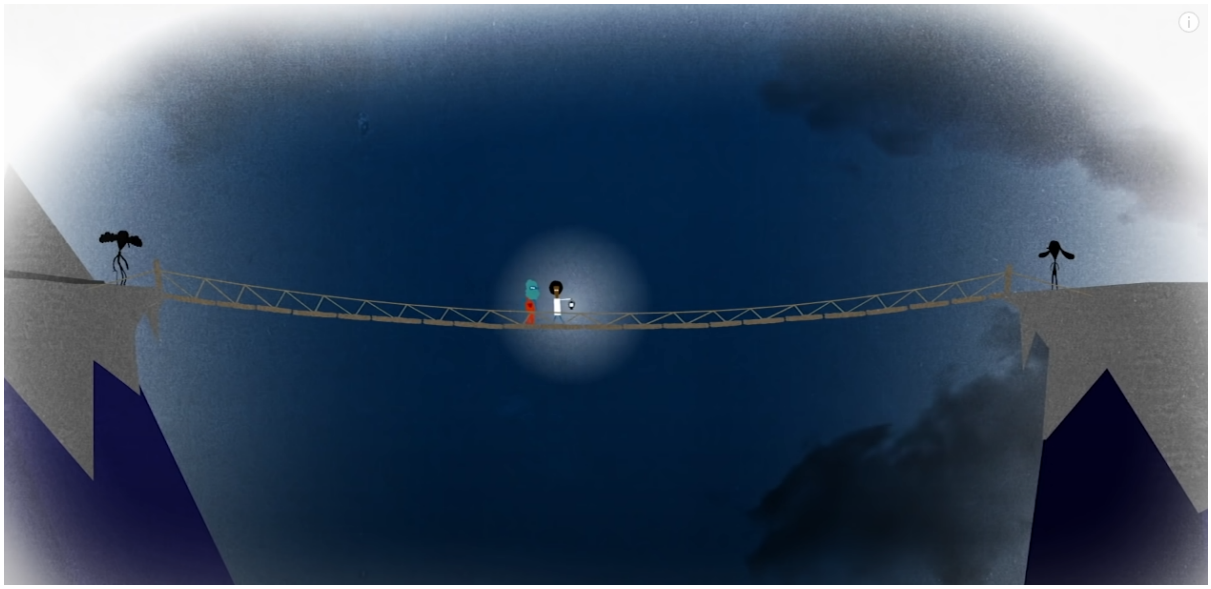
# Report

# 1. Presentation of the subject



"*Given 4 people, everybody at the left side of an old bridge wants to go to the right side, each with an independent time, with time requiring for a person is specified as a positive integer. But the problem is, the bridge can only hold up to 2 people, and while walking on the bridge, they must carry a candle, otherwise they cannot see the path, that leads to the fact that two people must go together if they both want to go through the bridge at the same time and one person have to go back with the candle. We know that the candle can go out at any time, so they need to all pass the bridge in the smallest time in order to prevent the risk of getting nowhere.*"

⇒ The goal is to take everyone from the left side to right side in **the minimum time.**

However, the upper subject is just the basic case. Our true subject is this subject with $n$ people randomly spawned on both sides of the bridge initially.

# 2. Description of the problem

## 2.1. Modify bridge crossing problem

We add more distinct attributes to the original problem:

- Instead of 4 people, now we consider $n$ people with walk time and random initial state for each.
- Restrict the problem with the new action set that allow only two 2 people cross the bridge at once and only 1 people crosses back.

We will prove that the second modification does not affect the optimal solution, which means the optimal solution for the restricted problem is also the optimal one for the original n-people Bridge and Torch problem.

**Proof**. The proof *[1]* works by an easy exchange argument.

Consider variants in which the solution deviates from the pattern $+\{x,x\}-x+\{x,x\}-x+\{x,x\}-$ ..., where $+\{x,x\}$ means two arbitrary people cross the bridge to the right side and $-x$ means one arbitrary person cross back to the left side.

- **Case 1**. The deviation is of the form $+a$. This cannot occur in first step, because otherwise the solution would have to begin with $+a-a$ , and these two steps are clearly redundant.

  Consider the move immediately before the offending move: $\cdots -b+a\cdots$ . The case a = b can be excluded. The last previous step in which a or b was moved is of the form $+\{b,c\}$ or $-a$. In either case, we can transform the solution to a faster solution as follows:

  $\cdots +b,c-\cdots -b+a\cdots \Rightarrow \cdots +a,c-\cdots \emptyset\emptyset\cdots$,
  $\cdots -a+\cdots -b+a\cdots \Rightarrow \cdots -b+\cdots \emptyset\emptyset\cdots$ ,
  with $\emptyset\emptyset$ indicating the two moves − b + a that were cancelled.

- **Case 2**. If the deviation is of the form $-\{a,b\}$, consider the last previous step in which a was moved. Let this be a move $+\{a,x\}$ (where $x=b$ is permitted). We can that cancel $a$ from both moves without increasing the total time:

  $\cdots +a,x-\cdots -a,b+\cdots \Rightarrow \ \cdots +x-\cdots -b+\cdots$,
  but the latter solution cannot be optimal, by the analysis of Case 1.

## 2.2. Task environment

Technically speaking, this is a problem solving by searching with properties of environment: *known, fully observable, deterministic, discrete* and *semi-dynamic:*

- Known: The new state, time,... are known when a cross is made for some people.
- Fully observable: Know all aspects relevant to the choice of action, which is the current position of everyone, walk time of each individual and the current position of the candle,...
- Deterministic: The next state of the environment is completely determined by the current state and the action.

- Discrete: A finite number of distinct states ($2^{n+1}$, included the candle) distinct states, where $n$ is the number of people involved), and at any given state there are only a finite number of actions to choose from.
- Semi-dynamic: The environment itself does not change with the passage of time but as the time passes, the performance score decreases.

Problem formulation:

- **Initial state**: The state of $n$ people as well as the candle is randomized initially, while still maintain the feasibility of problem (case like everybody at the left side and the candle at the right side is not allowed).
- **Actions/ Transition model**: 2 people go forward (if possible) and only 1 goes backward. The state of any entities (including people and candle) is represented as 0 or 1, where 0 is the original side (right) and 1 is the goal side (left).
- **Goal test**: Everyone is on the right side (side 1) of the bridge regardless of the position of the candle.
- **Path cost**: The time to cross the bridge of the slowest person walking the bridge at that moment.

Problem specification:

- Maximum branching factor $b$: $b = C_n^2$ since in the worst case, everyone is on the right side and 2 people can be randomly chosen to make a cross.

# 3. Choice of Algorithms

Among hundreds of search algorithms, we initially pick the two algorithms Uniform Cost Search and A* Search for the implementation whose potential can be seen intuitively. Consequently, some more algorithms has been put into practice, including Iterative Deepening A* Search, Branch and Bound, Depth First Search and Breath First Search.

## 3.1. Uniform Cost Search, A* Search and heuristics

*@graph search implementation*

*The reason behind these immediate selections is the optimality and familiarity as well*.

About Uniform Cost Search, since all the step costs, which are the durations of each person to cross the bridge, are positive integers, then one can theoretically conclude

that the strategy is optimal. Consequently, we expect it will perform optimally in practice.

About A* Search, the optimality is also guaranteed under consistent heuristics as the state space of the problem Bridge and Torch is finite ($2^{n+1}$) and the repeated states are avoided. Similarly, the expectation is therefore an optimally practical performance from A* Search in case we could invent a good-enough heuristic.

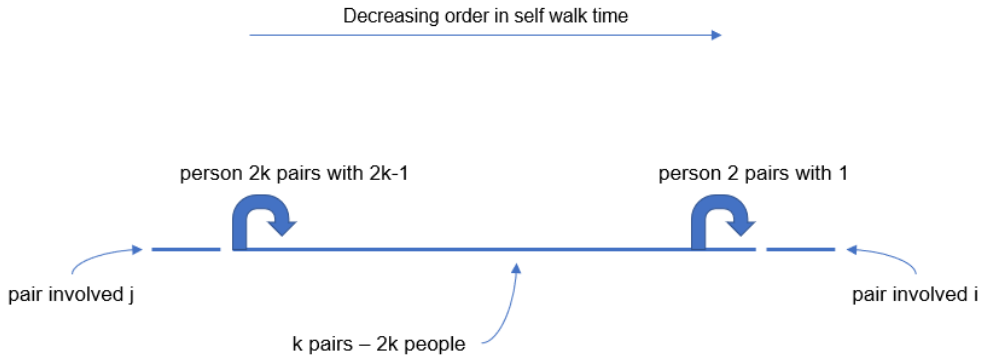
Some of our members have early proposed a variety of possible heuristics but at last, only two heuristics remain. The first one can be considered as a whole-relaxed variant while the second one is half-relaxed problem.

- The whole-relaxed heuristic: The original problem restricts the capacity of the bridge (maximum 2 people at once) and requires that people must cross the bridge with a torch. Now, the idea is to relax both of these constraints, then everyone just needs to cross the bridge normally, and the minimum time to bring all people to the goal side is the maximum time it takes for the slowest person on the original side to cross the bridge.
  Take a quick example: The current state is $[0, 0, 0, 1, 1, 1, 1]$ with the relative walk time being$[1, 2, 5, 7, 9, 8]$. Then, the value of heuristic is $5$.
- The half-relaxed heuristic: Instead of the full relaxation, only the torch is neglected. Thus, the value of the second heuristic will be the sum of the time taken by pairs (ordered walk time) to cross the bridge.
  An example is: The current state $[0, 0, 0, 1, 1, 1, 1]$ and the relative walk time $[1, 2, 5, 7, 9, 8]$. The remaining people in the original side is 1, 2 and 3 with their durations to cross the bridge is $1$, $2$ and $5$ respectively. First, we need to sort the list of these durations (thankfully, it has been sorted already), then sum up as follows: the 5-minute person pairs with 2-minute's to cross and 1-minute's goes alone. Then the value for heuristic is $5 + 1 = 6$.

Our reason to keep these two heuristics is due to their consistency. Indeed, our team has explored the hold for the consistency property in both of them and gradually completed the following math-based proof:

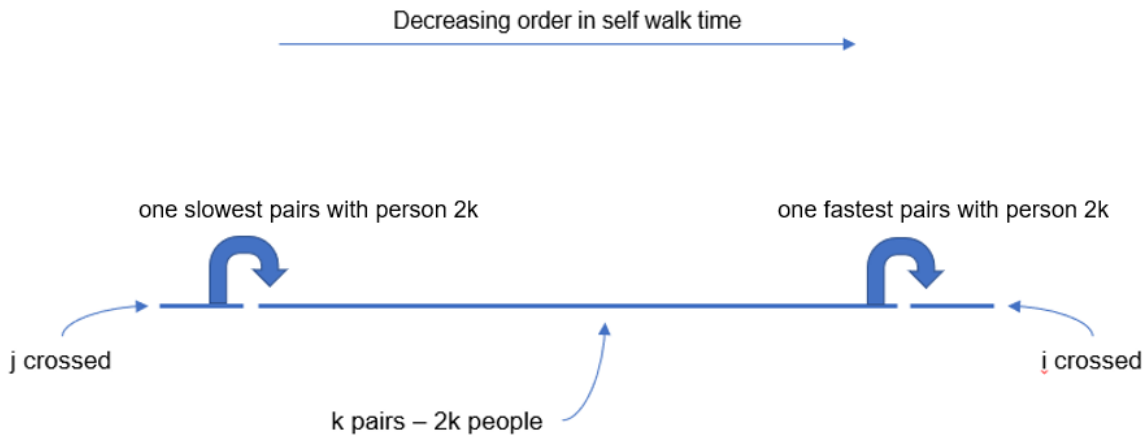
*Proof for the whole-relaxed heuristic*:
Recall: $h(n) \leq c(n, a, n') + h(n')$
$\Rightarrow\ h(n) - h(n') \leq c(n, a, n')$
- Not bring the fastest person on the original side: $h(n) = h(n')$
- Otherwise: $c(n, a, n') = h(n)$
$\Rightarrow -h(n') \leq 0$ (always true)
$\Rightarrow$ The whole-relaxed heuristic is consistent

*Proof for the half-relaxed heuristic*:
Without the loss of generality, the walk time list for the remaining people is assumed to be sorted. The action is now to make a cross for 2 people $i^{th}$ and $j^{th}$ ($i < j$) on the original side, so the step cost is definitely $c(n, a, n') = (j)$ with $(j)$ being the time for $j^{th}$ to cross the bridge alone.
Intuitively, one can see that such actions only shift the pairing backward 1 element for all elements in range from the pair involved $i^{th}$ to the pair involved $j^{th}$, which means the impacts is on the subsequence of that range solely. Therefore, the proof is now reduced into proving that if a cross is made for the first or second fastest person and first or second slowest one, then the inequality $h(n) \leq c(n, a, n') + h(n')$ holds.
The below illustration is for above subproblem we will prove right after:




Let's $i := \{1^{st}\ fastest, 2^{nd}\ fastest\}$ and $j := \{1^{st}\ slowest, 2^{nd}\ slowest\}$
$h(n) = (1^{st}\ slowest) + (2k) + (2k-2) + ... + (4) + (2) + (2^{nd} fastest)$
- If a cross is made for $j$ as the $1^{st}$ slowest and $i$ as any the fastest:
$\Rightarrow h(n') = (2^{nd}\ slowest) + (2k-1) + (2k-3) + ... + (3) + (1)$
$\Rightarrow c(n, a, n') = (1^{st} slowest)$
$\Rightarrow$ The inequality $h(n) \leq c(n, a, n') + h(n')$ is:
$(2k) + (2k-2) + ... + (2) + (2^{nd} fastest) \leq (2^{nd}\ slowest) + (2k-1) + (2k-3) + ... + (3) + (1)$ (always true)

$\Rightarrow$ Consistency holds
The proof the for the remaining case ($j$ as the $2^{nd}$ slowest) is an analogue. Below is the illustration after $i^{th}$ and $j^{th}$ moved.




For the action in which a back cross is made for one person from goal side to the another, the value of heuristic obviously increases or equal to the previous, hence $h(n) - h(n') \leq 0$ and then the inequality holds.

Knowing that both heuristics are consistent grants us a chance to explore further about dominance. Definitely, the value of the half-relaxed heuristic is higher than that of the whole-relaxed, leading us to a conclusion that the half-relaxed dominates the another. Consequently, we expect half-relaxed heuristic-used algorithms will perform better than whole-relaxed heuristic-used do in practice.

As the heuristic observation is done with A* optimality being guaranteed theoretically, the discussion can continues to time complexity, space complexity and running time. The fact is that A* Search is optimal efficiency *[2]*, that means among all optimal strategies that start from the same start node and use the same heuristic $h$, A* Search expands the minimal number of paths, so provided heuristics used are high-quality, A* Search is expected to outperform Uniform Cost Search and the other optimal strategies in most aspects during experiments.

## 3.2. Iterative Deepening A* Search

*@tree search implementation*

Iterative Deepening A* Search is a variant of A* Search (Memory-bounded heuristic search), which saves a huge amount of memory but acts horribly slowly in return due to re-expanding many nodes. *Our reason to implement it is therefore to observe these prominent properties in experiments, making a comparison to A* Search*.

The expectation here is that this strategy will take less memory than any other optimal algorithms applied in the project but struggle hard in time. In addition, because the conditions for A* Search optimality is satisfied under our two heuristics,

Iterative Deepening A* Search is optimal theoretically. In experiments, we then expect for its optimality.

## 3.3. Branch and Bound

*@advanced implementation*

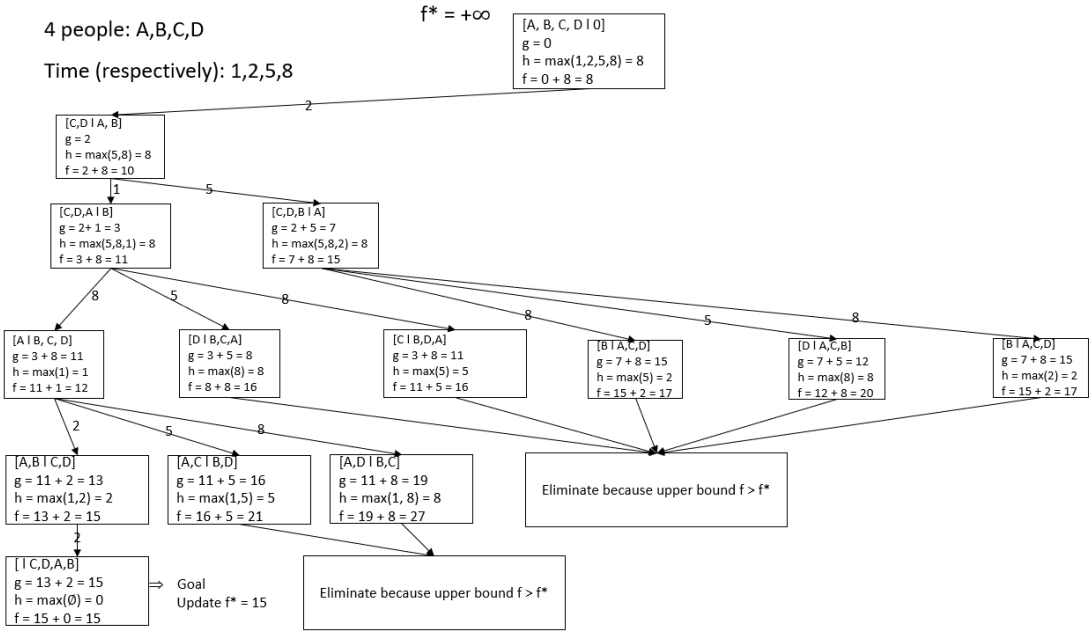
A strategy comes from outside of the subject area, Branch and Bound, which is also a memory-saving strategy but believed to *perform faster than Iterative Deepening A* Search*.

Branch and Bound is an algorithm design for discrete and combinatorial optimization problems, as well as mathematical optimization. A branch-and-bound algorithm consists of a systematic enumeration of candidate solutions by means of state space search: the set of candidate solutions is thought of as forming a rooted tree with the full set at the root. The algorithm explores *branches* of this tree, which represent subsets of the solution set. Before enumerating the candidate solutions of a branch, the branch is checked against upper and lower estimated *bounds* on the optimal solution and is discarded if it cannot produce a better solution than the best one found so far by the algorithm.

The algorithm depends on efficient estimation of the lower and upper bounds of regions/branches of the search space. If no bounds are available, it degenerates to an exhaustive search. *[3]*

In this problem, Branch and Bound is collaborating with the whole-relaxed heuristics. Since the heuristics have been proved to be consistent and dominant, it can play as the closest lower bound functions for the cost function.

As the objection is always feasible, Branch and Bound will go straight down till it reaches the goal, update its optimal cost function, then backtrack to node which can still expand. Because the algorithm check lower bound of expanded node whenever it expand, after searching the whole tree, we will always obtain the optimal solution.




There you see, every branch that have lower bound greater than $f$-optimal will be cut-off. So in general, the better initial state we consider, the outstanding result it might be. To summarize, branch and bound is a snappy solution for this problem.

## 3.4. Depth First Search and Breadth First Search

*@graph search implementation*

Depth First Search and Breadth First Search are the only two non-optimal algorithms, but they are complete since the repeated states are avoided. That means a feasible schedule is always found for any valid instances using these strategies.

The property that no repeated states are re-explored guaranteed that there will be a solution for a valid instance after $2^n$ nodes are expanded and, based on strategies themselves, their solution can be either optimal or feasible only. Therefore, the prediction is that the time complexity of these two strategies are around that of Uniform Cost Search and A* Search. However, the priority queue implementation of those optimal algorithms possibly increases their running time significantly, compared to that of Breadth First Search and Depth First Search. *This means the analysis on this couple could gain us an insight into the quality of implementation*.

# 4. Implementation

Following general graph search algorithm, we have implemented four different search strategies A* Search, Uniform Cost Search, Breadth First Search and Depth First Search.

```
fuction Graph-Search(problem, fringe) returns a solution, or failure
  fringe <- Insert(Make-Node(Initial-State(problem)), fringe);
  closed <- an empty set;
  while (fringe not empty) {
    node <- RemoveFirst(fringe);
    if (Goal-Test(problem, State(node))) then return Solution(node);
    if (State(node) is not in closed) then
      add State(node) to closed
      fringe <- InsertAll(Expand(node, problem), fringe);
    end if
  end
  return failure
```

*General graph search pseudocode [4]*

Iterative Deepening A* Search is a tree search algorithm, combining Iterative Deepening Search and A* Search, therefore the way we implement this strategy is a little bit different:

```
procedure ida_star(root)
    bound := h(root)
    path := [root]
    loop
        t := search(path, 0, bound)
        if t = FOUND then return (path, bound)
        if t = ∞ then return NOT_FOUND
        bound := t
    end loop
end procedure

function search(path, g, bound)
    node := path.last
    f := g + h(node)
    if f > bound then return f
    if is_goal(node) then return FOUND
    min := ∞
    for succ in successors(node) do
        if succ not in path then
            path.push(succ)
            t := search(path, g + cost(node, succ), bound)
            if t = FOUND then return FOUND
            if t < min then min := t
            path.pop()
        end if
    end for
    return min
end function
```

*Iterative Deepening A* Search pseudocode [5]*

Branch and Bound is an algorithm that is built different. Generally, the idea of branch and bound is almost close to brute force, a kind of smarter brute force, so its design is the combination of backtracking algorithm and a lower-bound condition. Consequently, we consider it as a tree search implementation.

```
procedure branch_bound(problem, root):
  fx <- INF
  x <- None
  if Goal-Test(problem, State(root)) then return Solution(root)
  branch(problem, root)
  if fx <- INF then return failure
  return Solution(x)

fuction branch(problem, node):
  nonlocal fx, x
  for succ in Expand(node, problem):
    if Goal-Test(problem, State(succ)) and succ.f < fx:
      fx = f
      x = succ
    else
      if succ.f < fx:
        branch(succ)
  return
```

*Branch and Bound pseudocode*

From pseudocodes above, one noticed that they share some functions such as `Expand()`, `Solution()`, `Goal-Test()` ,... and objects like `problem` , `node` ,... Hence, we come up with an idea that is to build a structure consisting of a class which models everything of the problem and a class which contains search algorithms and take the former class object as a parameter to provide rules and objectives for searching. Then, at last, we have developed classes:

- `Problem()` → `BridgeTorch(durations, init_state)` : The model of Bridge and Torch problem, with `durations` being a string of walk time of each person and `init_state` a binary sequence whose element is a position of each person (0: original side, 1: goal side) and last element is the position of the candle at that moment.
- `Node(problem, state, parent, action)` : The node object used in searching, with `problem` being the model of a Bridge and Torch instance, `state` current state, `parent` the reference to the parent node of the current one and `action` the single action that make the parent's state current state.

- `GraphSearch(problem)` : Construct a search graph using any strategies built in as methods (e.g. Uniform Cost Search), with `problem` being the model of a Bridge and Torch instance we need to solve.
- `TreeSearch(problem)` : Inherit `GraphSearch(problem)` but construct a search tree instead by using any strategies (e.g. Iterative Deepening A* Search) built in as methods.

## 4.1. Problem environment in programming

Initially, Bridge and Torch problem needs to be reconstructed in programming, for which class `BridgeTorch(durations, init_state, objective=None)` comes on duty.

In order to rebuild the problem characteristics, this class holds 4 distinct attributes:

- `self.durations` ← `List(duration)` (explained above)
- `self.init_state` ← `List(init_state)` (explained above)
- `self.goal` : The goal state in which everyone reached the side 1 (goal side)
- `self.objective` ← `objective` : The minimal value of objective of an instance. It is inserted in case we have already known the optimal total duration to cross the bridge for that instance.

Have defined the attributes, we develop method `SuccessorFn(Node.state)`, the transition model of Bridge and Torch in program. Simply, this method is built based on the problem formulation in 2. Description of the problem. During the development process, we found out a special case that have not been covered yet; it is the state when there are only one person left with the candle being in the side 0 (original side). Since we have limited the set of actions in which exactly two people cross the bridge, then our program are unable to find any solution for such cases even though it is just an easy candy. Therefore, a piece of script has been written by our programmers to specify this unexpected situation.

One more thing to talk about `SuccessorFn(Node.state)` is the way to express a transition, or a set of action and resultant state in other words. For convenience, action is of the form: `[[person_x, person_y], (origin, goal)]` with person $x$, person $y$ being the two people will make a cross and origin and goal the side they start and the side they come respectively. In case of just one individual chosen to cross, the form is `[[person_z], (origin, goal)]` relatively.

The transition model is definitely of importance, but it is not up to everything. The problem also requires distinct step costs for each action, which means we have to

build a method to cover this part, that is `StepCost(Node.action)`. Building it is not difficult since Bridge and Torch states that the cost for each action is the time to cross the bridge of the slowest person chosen to walk the bridge at that moment.

There are some more methods developed for the problem model (e.g. `Solution(Node.state)`, `GoalTest(Node.state)`); and all of them are noted in module **Search.py**.

## 4.2. Search engine and measurements

Generally, one indispensable material for searching is the object node. The class `Node(problem, state, parent, action)` is hence constructed first, opening up the idea about the two search classes `GraphSearch(problem)` and `TreeSearch(problem)`.

The purpose of the couple `GraphSearch(problem)` and `TreeSearch(problem)` is to deploy graph or tree search algorithm using any strategies developed as their methods. Early, our ways to implement search strategies were mentioned using pseudocode, then one issue left is the function, or say: Method `Expand(Node)`.

Developing directly such methods seems to be exhausted and ends up messy, but because we have already built the method `SuccessorFn(Node.state)`, things turn to be simple so far. The short pseudocode below will explain the expansion procedure:

```
fuction Expand(node, problem) returns a set of nodes
  successors <- empty set
  for each action, result in problem.SuccessorFn(node.state) do
    s <- a new node
    s.parent <- node; s.action <- action, s.state <- result
    s.path-cost <- node.path-cost + problem.StepCost(action)
    s.depth <- node.depth + 1
    add s to successors
  return successors
```

*Expanding a node in pseudocode [6]*

The program is now set up greatly, and able to solve any problem instances as expected. Nonetheless, it is uncompleted if there is any demands in analysis of algorithm performance. The team therefore have set up some parameters in the middle of search strategies to measure their time complexity, space complexity and running time.

Some schemes have came on the stage, but personally the most promising idea is to create `self.time_complexity` and `self.space_complexity` as the search classes'

attributes and put them in method `Expand(Node)`, then each time a node is generated, each parameter will add 1 to the value themselves.

```
program GraphSearch(problem)
  <...>
  fuction Expand(node, self) returns a set of nodes
    successors <- empty set
    for each action, result in self.problem.SuccessorFn(node.state) do
      <create node> ...
      self.time_complexity = self.time_complexity + 1
      self.space_complexity = self.space_complexity + 1
      add s to successors
    return successors
```

Nodes are not kept in memory forever, usually they are gone when we pop them out of fringe. However, a node is, except for the root node, kept a pointer to its parent, that means even though a node has been popped, it still can stay in memory as long as its children are in memory. Following this, space complexity will decreases only if current leave nodes, which are not goal nodes, are popped but not for expanded.

Furthermore, we recognised that the measurement method built for space complexity would provide just the number of node kept in memory at the end of searching procedure so far. Thus, one more attribute (`self.max_space_complexity`) has been added in order to record the maximal number of nodes in memory right before the system deletes any nodes.

For running time, built-in module `time` is enough to handle, and everything is now ready to be put on analysis table.

# 5. Comparing the results of the algorithms

The number of random test is massive, hence module **BackTest.py** has been built to execute the benchmark and collect data about the performance of each algorithm into .csv files in folder **performance** for later analysis.

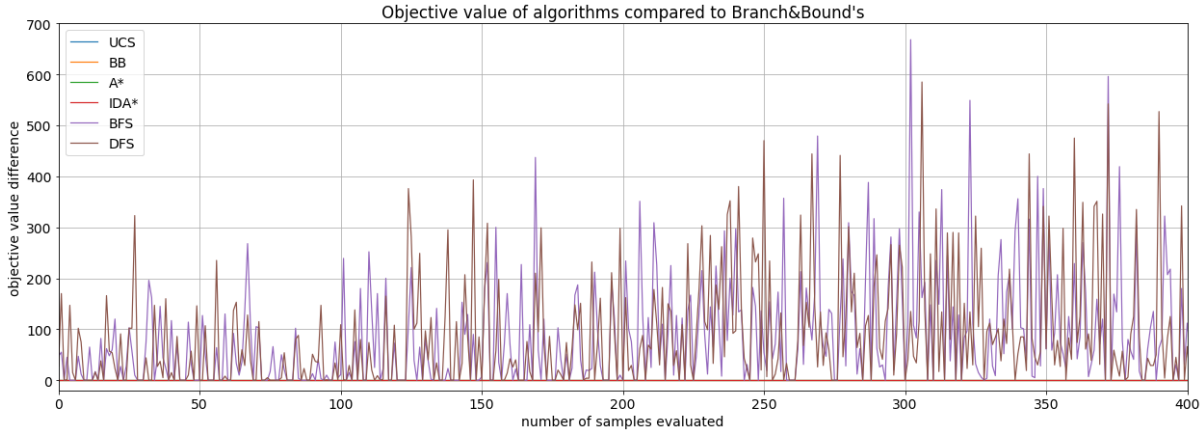
Since the the half-relaxed heuristic dominates the another, we benchmark heuristic strategies using the former one.

## 5.1. Optimality

All of algorithms, except for Breadth First Search and Depth First Search, are proven to be optimal, so their test result should return the same objective value for each sample and vice versa in case of non-optimal algorithms. To analyse easily, Branch

and Bound’s objective value are chosen to be a pivot for the other, and if there are any objective values for a random sample included in the test whose subtraction to Branch and Bound’s objective value is greater than 0, they should be either Breadth First Search’s or Depth First Search’s. Nonetheless, it might be errors in our implementation or analysis.

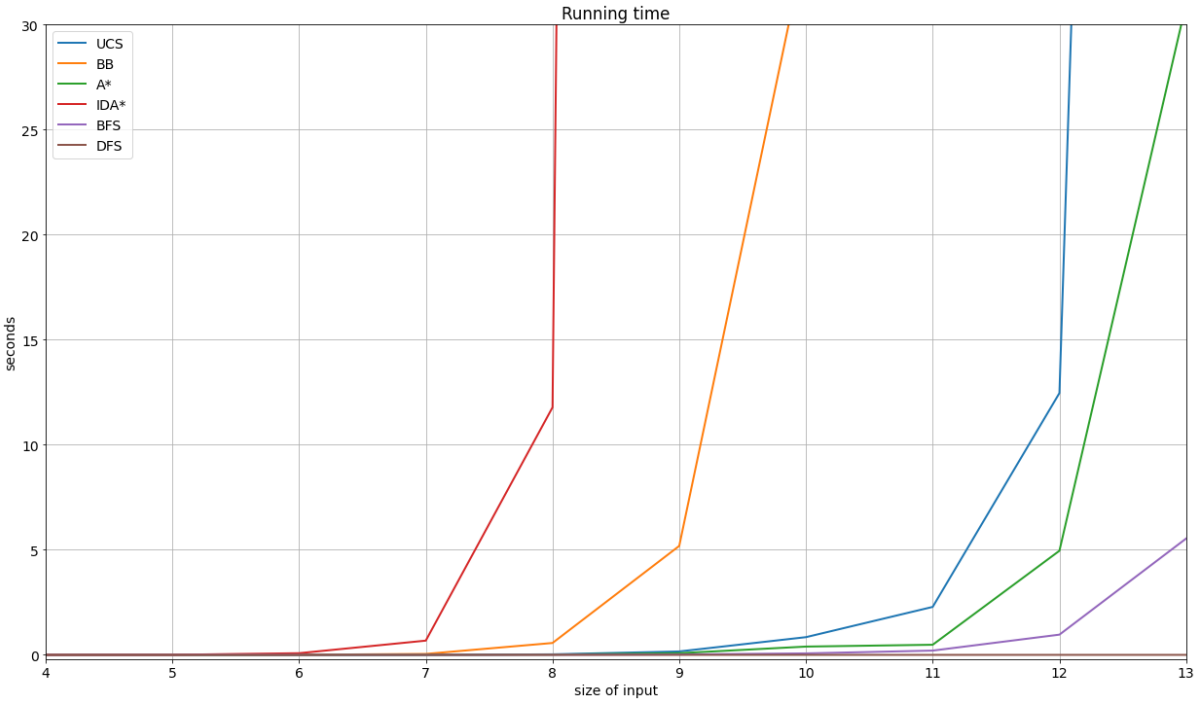
We have done 400 random tests (from size 4 to size 7 of inputs) for the five algorithms applied during the work. Below is the graph describing our observation visually.




The graph shows that all of the optimal algorithms’ objective value line are horizontal ($y = 0$) except for Breadth First Search’s and Depth First Search’s since we can see they vary a lot from the optimal line. Quantitatively from the dataset, both Breath First Search and Depth First Search could solve just 126 instances optimally. The other algorithms, on the other hand, solved 400 instances optimally over 400, which is as our expectation.

## 5.2. Running time and time complexity

In this section, there are 100 random tests carried related to each size of input in range from 4 to 7 and 50 random tests did in range from 9 to 13, because from the size 14 onwards, all of algorithms take averagely over 30 minutes to solve a problem instance.

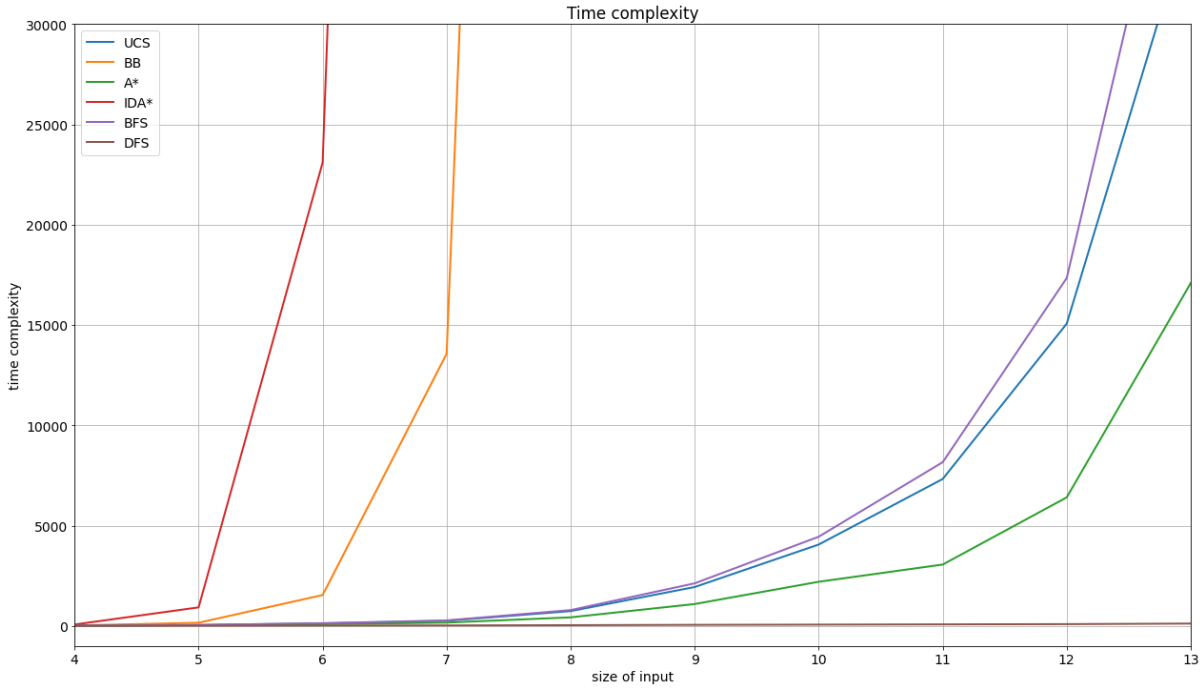



*Average running time*

As the graph shown, Iterative Deepening A* Search is the slowest algorithm since its running time soon rockets just at the size 9. Branch and Bound also struggles to solve instances of size larger than 10 although its performance is significantly better than Iterative Deepening A* Search at the size 9. These are the result from re-expanding too many nodes and brute force-like behaviour, respectively.

On the other hand, the couple, Uniform Cost Search and A* Search, suffer less from the increase in size of inputs. However, it seems that their limit is the size 13 where their wall-clock time surges dramatically. Furthermore, one is interesting that, A* Search considerably outperforms Uniform Cost Search, which means we have invented a good-enough heuristic.

Breadth First Search and Depth First Search are even faster so far, which is not our expectation that their running time approximates A* Search and Uniform Cost Search. However, we cannot jump to any conclusion here since the cause is possibly the poor implementation of priority queue as mentioned in theoretical analysis and hence an insight into time complexity probably helps us.

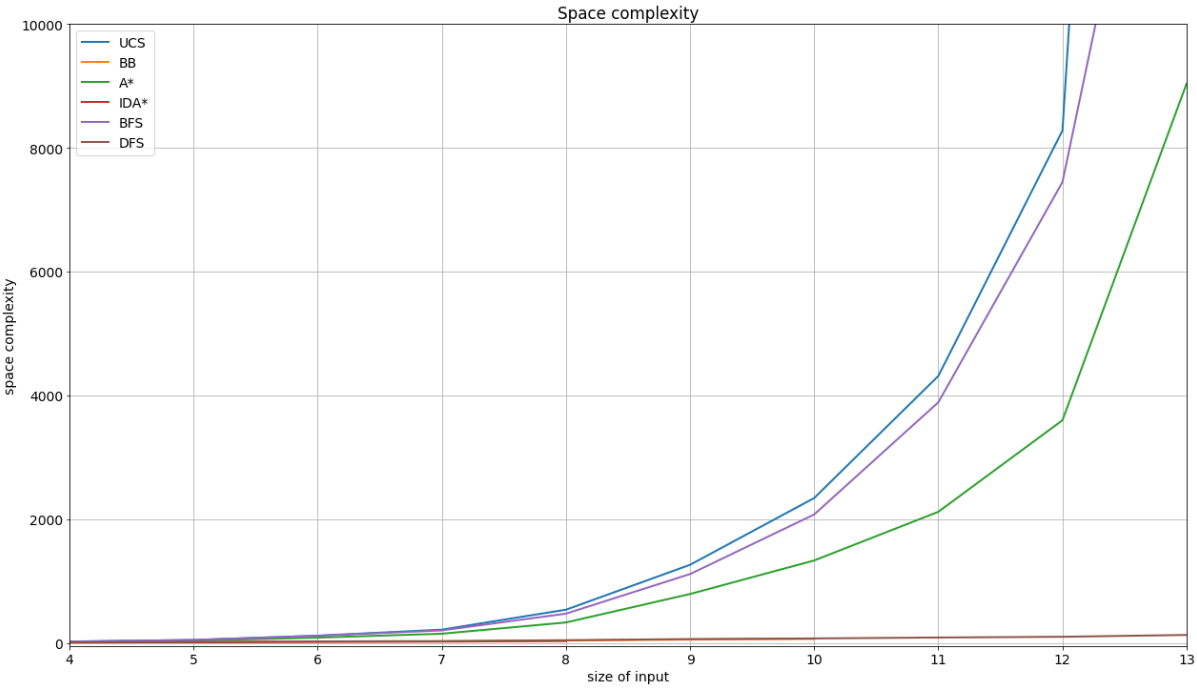



*Average time complexity*

The second graph shows that the number of nodes generated by Breadth First Search is higher than both Uniform Cost Search and A* Search. Consequently, one can say that Breath First Search is actually slower in theory but faster in practice due to the quality of implementation.

Everything now follows our expectation, except for Depth-First Search, which remains its lightning speed in both charts regardless of the size of input. Advanced observation shows that Depth First Search just generates around 260,000 nodes averagely in size 100, surprising us so far. In our opinion, the reason is that Breadth First Search visits all the nodes on the same layer, but Depth First Search always finds at least one child node of any parent node to expand, which shortly means it just expand 1 node in each level of the search space.

## 5.3. Space complexity




*Average space complexity*

In term of space complexity, no surprise appeared. A* Search with its efficient heuristic dominates the two Uninformed Search strategies, Breadth First Search and Uniform Cost Search, as expected. Branch and Bound and Iterative Deepening A* Search take approximately the same amount of memory, whose is the smallest compared to the other. Depth First Search also performs greatly in term of space, which is understandable.

# 6. Conclusion and possible extensions

## 6.1. Conclusion

Upon solving Bridge and Torch problem, some conclusion can be drawn in regard to the theoretical and practical analysis:

- In term of time: A* Search (the half-relaxed heuristic) works best, but if we just want a feasible solution, Depth First Search will stand out.
- In term of space: Branch and Bound shows its domination since it saves as much space as Iterative Deepening A* Search does while its growth rate in time is significantly smaller than that of the another. Depth First Search is also promising in case no demand in optimality.

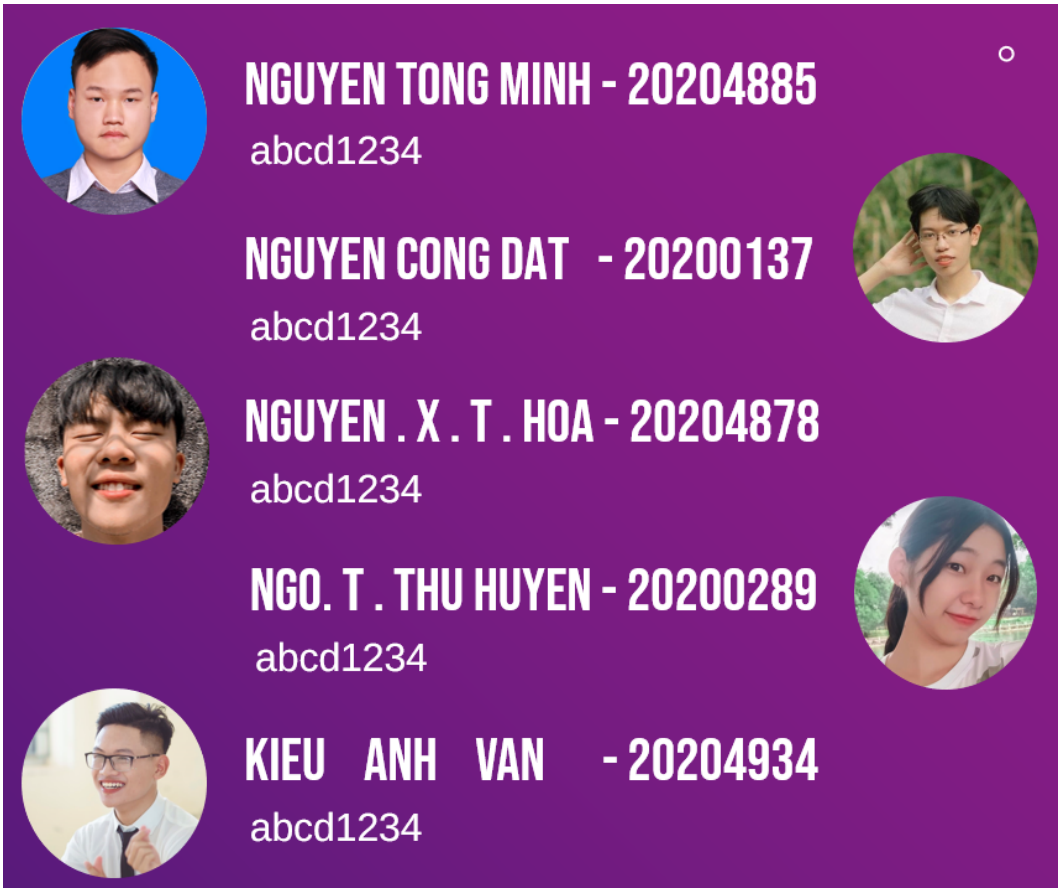
Generally, all of the algorithms applied, except for Depth First Search, takes more than 30 minutes to solve average instances of size larger than 13.

## 6.2. Possible extensions

Some possible extensions follows:

- Better implementation of priority queue: List is not suitable for implementing priority queue since finding the appropriate element and pop it (comes with re-indexing) require $O(l)$ ,with $l$ being the length of the fringe at that moment, for each. Therefore, interactions with fringe are expensive, especially when the fringe contains thousands of nodes.
    - Possible measure: Heap queue implementation (built-in module `heapq`). Particularly, a min binary heap supports `insert()`, `delete()` and `extractmin()` operations in $O(logn)$ time with $n$ being the length of heap *[7]*.

# 7. List of tasks




Group 11, Members

## 7.1. Programming task

**Algorithm implementation**:

- Implementing Breadth First Search & Depth First Search: Ngô Thị Thu Huyền
- Implementing Uniform Cost Search: Nguyễn Tống Minh
- Implementing A* Search: Nguyễn Xuân Thái Hòa
- Implementing Iterative Deepening A* Search & Branch and Bound: Nguyễn Công Đạt

**Subject-related implementation**: Nguyễn Tống Minh

- Construct problem environment (class `Problem`)
- Construct call object (class `Solver` - shorten process to call methods and used in benchmark)
- Construct search engine (class `GraphSearch`, class `TreeSearch`, class `Node`)

**Testing and debug**:

- Benchmark algorithms (module `BackTest.py`, performance dataset, graph sketch using Matplotlib & Pandas): Nguyễn Tống Minh
- Debug:
  - Nguyễn Xuân Thái Hòa: Detect the malfunction in the method `Heuristic2`(half-relaxed) of class `Problem` due to false list-type index assignment, that causes the value of the half-relaxed heuristic is always 0.
  - Kiều Anh Văn: Fix the error in module `BackTest.py` that causes data written in wrong files

## 7.2. Analytic tasks

**Report writing**:

1. Presentation of the subject: Nguyễn Công Đạt
2. Description of the problem: Nguyễn Công Đạt
3. Choice of algorithms: Nguyễn Tống Minh, Nguyễn Công Đạt *(3.3. Branch and Bound)*
4. Implementation: Nguyễn Tống Minh, Kiều Anh Văn *(4.1. Problem environment in programming)*

5. Comparing the results of the algorithms: Nguyễn Xuân Thái Hòa (50%), Ngô Thị Thu Huyền (50%)
6. Conclusion and possible extensions: Nguyễn Tống Minh *(6.2. Possible extensions)*, Kiều Anh Văn *(6.1. Conclusion)*
7. List of tasks: Nguyễn Công Đạt, Nguyễn Tống Minh
8. List of bibliographic references: Nguyễn Tống Minh

**Powerpoint & Demo**: Kiều Anh Văn

**Task Assignment**: Nguyễn Xuân Thái Hòa, Nguyễn Tống Minh

**Subject proposal**: Nguyễn Xuân Thái Hòa

# 8. List of bibliographic references


[1] Rote, Günter. Crossing the Bridge at Night. Bulletin of the EATCS. 78. 241-. https://page.mi.fu-berlin.de/rote/Papers/pdf/Crossing+the+bridge+at+night.pdf, A3:11-A3:33.

[2] Artificial Intelligence – A Modern Approach. Stuart Russell and Peter Norvig. Prentice Hall, THIRD EDITION, pages 98.

[3] Wikipedia. Branch and Bound. https://en.wikipedia.org/wiki/Branch_and_bound

[4] Muriel Visani, Le Thanh Huong and Tran Duc Khanh. Introduction to Artificial Intelligence IT3160E, Chapter 3 - Problem solving, Part 2: Problem-solving by searching using informed search strategies (2021), page 8.

[5] Wikipedia. Iterative deepening A*. https://en.wikipedia.org/wiki/Iterative_deepening_A*

[6] Muriel Visani, Le Thanh Huong and Tran Duc Khanh. Introduction to Artificial Intelligence IT3160E, Chapter 3 - Problem solving, Part 1: Problem-solving by searching (2021), page 38.

[7] GeeksforGeeks. Binary Heap. https://www.geeksforgeeks.org/binary-heap/