HO CHI MINH UNIVERSITY OF TECHNOLOGY AND EDUCATION

FACULTY FOR HIGH QUALITY TRAINING

Final Report

ARTIFICIAL INTELLIGENCE

Lecturer: PhD. Tran Nhat Quang

Class: ARIN330585E_22_1_01CLC

Subject ID: 221ARIN330585E

Members:

Nguyen Minh Tri – 20110422

Phan Thanh Luan – 20110380

Tran Minh Gia Khanh – 20110374

Le Diep Tri - 20110421

Thu Duc city, December 4th, 2022

TABLE OF CONTENTS

INTRODUCTION

We would like to express our gratitude to the professor, PhD. Tran Nhat Quang, who helped us personally throughout the topic-making process in order to successfully complete this topic and this report. We appreciate the teacher's advise from his real-world experience in helping us meet the requirements of the chosen topic, as well as his willingness to always respond to our queries and offer suggestions and corrections. time to assist us in overcoming our flaws and completing it successfully and on time.

We also want to extend our sincere gratitude to the instructors in the High Quality Education Division generally and the Information Technology sector specifically for their committed expertise that has helped us build a foundation. The conditions for learning and performing effectively on the topic have been created by this topic. We also want to express our gratitude to our classmates for sharing expertise and insights that helped us refine our topic. We created the subject and report in a little amount of time, with little expertise and numerous other technical and software project implementation difficulties. Therefore, as it is inevitable that a topic may have flaws, we eagerly await the lecturers' insightful comments in order to further our knowledge and improve for the next time.

We appreciate you very much.

Finally, we would want to wish all of you teachers, ladies and gentlemen, continued health and success in your line of work with developing individuals. Again, we appreciate your kind words.

We sincerely thank you.

TASK ASSIGNMENT

Student's name	Evaluate contribution	Taskwork
Nguyễn Minh Trí Leader	100%	Making content(V) 8 -> 10. Words Interface. Coding the Application User Interface.
Trần Minh Gia Khánh	100%	Making content from (IV) 1 -> 6.5. Coding the project (A Star).
Phan Thành Luân	100%	Making content from (V) 4 -> 7. Coding the project (A Star).
Lê Diệp Trí	100%	Making content form (IV) 7 -> 3 (V). Coding the project (IDS).

Note: %: The percentage of each student participating.

No	Goal	Schedule						
1	Understand & describe the requirements of the project.	o	o					
2	Learn & study technology, language programming to do project	o	o	o				
3	Identify the functions and algorithm to be	o	o	o			o	

	used in the project and report.							
4	Building & design project architecture.	o	o	o				
5	Building(Coding) & design content of the project and report architecture.			o	o	o		
6	Testing the project.					o	o	o
7	Write report.					o	o	o
Week		2	4	6	8	10	12	14
Note	o-Begin	O – Complete 50%				O – Complete 100%		

Remark of teacher:

.....

.....

.....

.....

.....

.....

.....

.....

Thu Duc city, December 4th, 2022

Signature and full name of teacher

I. PLAGIARIZE, PLAGIARISM.

1. What is plagiarism?

- Plagiarism is an academically dishonest action, is a serious ethical violation, and can sometimes constitute copyright infringement.
- Manifestations of plagiarism:
    - Copy other people's ideas and words into your own.
    - Completely use other people's ideas and styles without crediting sources.
    - Sometimes, even if there are sources, too much copying can be considered plagiarism.

2. Dos/don'ts of reporting

- Do not use other people's words and code without crediting sources.
- Don't completely copy other people's ideas, code.
- Do your own research and complete the report as much as you understand it.
- Fully preparing theoretical and practical knowledges when reporting.

3. Commitment

We would like to assure you that this project was carried out by our team members. We take full responsibility If we commit plagiarism (such as using other people's documents and code without crediting sources, or copying more than 30% of the report).

II. LEARN ABOUT ARTIFICIAL INTELLIGENCE (AI)

1. What is Artificial Intelligence?

- Intelligence is understood as the ability to think, deduct, recognize and learn
- Artificial is considered as anything made by humans that is not naturally occurring
- Understandably, Artificial Intelligence is a technology that simulates the processes of human learning such as recognizing, thinking, deducting, and learning for machines, especially computer systems.

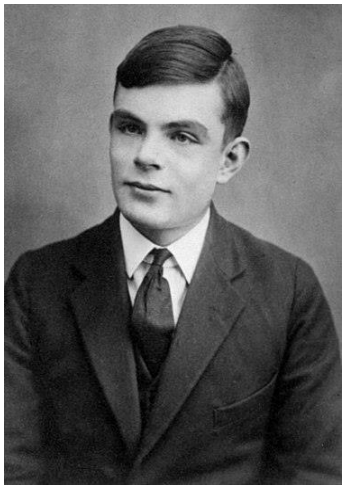
2. History of AI

2.1. Rossumovi Univerzální Roboti play

- In 1920, Karel Čapek ,a Czech writer, produced a science fiction play called Rossumovi Univerzální Roboti (Rossum's Universal Robots, abbreviated as R.U.R). This is an interesting play and it is considered the first milestone in the history of AI.
- The play tells the story of a factory that uses organic materials to make artificial humans called robots. These robots differ from today's definition of robotics, they are made of flesh and blood, not machines. At first, robots were a tool that helped humans a lot at work. But then they rebelled and led to human extinction.
- Although the robot idea in the play is not the same with the idea of robots today ,it introduced the term robot and some kind of intelligence man-made is both useful to humans and a threat to humans if they are not controlled.
- Artificial Intelligence in Art is a very different topic but it influences the study of Artificial Intelligence in science and its application in life.

2.2. Alan Turing and the test

- Alan Turing is a British mathematician, logician and cryptographer, he is regarded as the father of Computer Science in general and Artificial Intelligence in particular.
- A few years after the Second World War, Alan Turing introduced a test (Turing test). The test was introduced in "Computing Machinery and Intelligence" when he was working at the University of Manchester in 1950.
- The Turing test is used to determine the intelligence of the machine. According to the test, a computer can be considered intelligent if the computer (A) and a person (B)



- Alan Turing (1912–1954)

concurrently communicate in natural language with another person (C), and (C) cannot distinguish between machines (A or B).

2.3. The term "artificial intelligence" and the Dartmouth conference (1955–1956)

- Along with Alan Turing, Marvin Minsky, Allen Newell, Herbert A. Simon, **John McCarthy** was one of the founding fathers of Artificial Intelligence . McCarthy named the term "artificial intelligence" in 1955, which was the main subject of the Dartmouth conference in the summer of 1956.
- The Dartmouth conference was probably the first workshop on Artificial Intelligence and formed the science of research on Artificial Intelligence.
- Researchers from Carnegie Mellon University, the Massachusetts Institute of Technology, and employees from IBM met each other at the conference and worked together on the creation of Artificial Intelligence research labs.
- A few years later, research on Artificial Intelligence achieved great successes. At this time, all experts in Artificial Intelligence are extremely optimistic about the future development of this technology.
- After that, Artificial Intelligence had some periods of stagnation such as the years 1970, 1987-1997.
- The 21st century has great potential to be the strongest development time of Artificial Intelligence with applications such as Big Data, Deep Learning and widespread commercial applications...

3. Application of Artificial Intelligence

Artificial intelligence is applied in many practical fields.

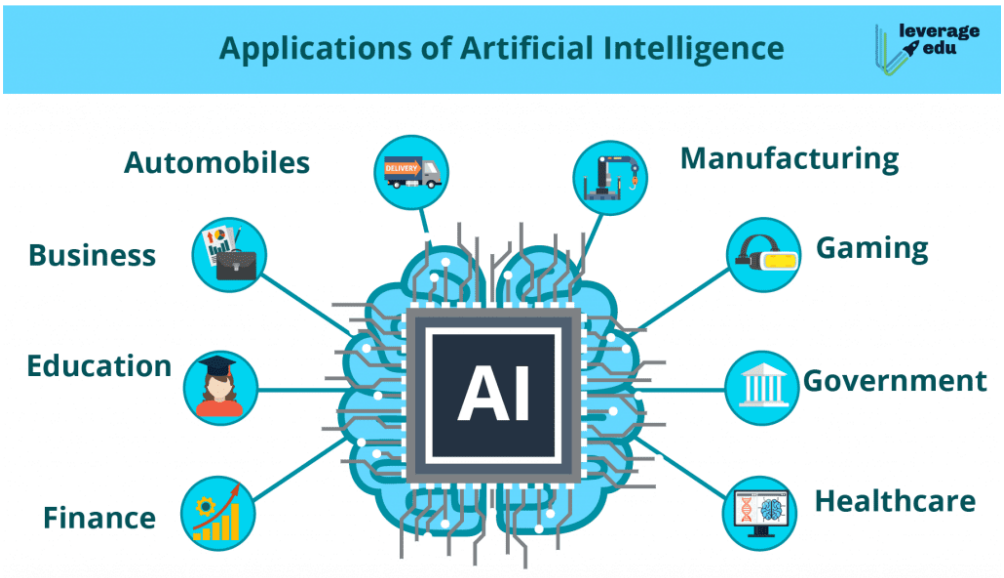


Image source : *https://leverageedu.com/blog/applications-of-artificial-intelligence/*

III. ARTICLE INTELLIGENCE ETHICS.

In the era of development, technology is gradually replacing human positions in every field. It is impossible not to mention the applications of AI to things that we cannot imagine. The sudden explosion of AI has brought a lot of benefits to humans, besides there are also bad sides. So is AI ethical or not?

Ethical Fields of AI:

Virtue ethics:

> Virtue is the most important thing of man, the source of other virtues. To be a man, they must cultivate virtue first.

Deontology:

> The field follows the rules. For this Ethical Field, people are considered good if they act according to the laws, and obey those laws.

Consequentialism:

> Determine the action whether it is bad or good by the results. Whether actions are considered good or bad will be based on the results it brings. If it is good, the action is right, otherwise it will be eliminated.

In conclusion, we can see that AI is a Artificial product, so it will be partly influenced by human ethics. Many scientists are still wondering about applying ethics to Artificial Intelligence tools. But if it doesn't, it could be an implicit threat to society and partly slow down the development of the AI industry.

IV. SEARCH ALGORITHMS

1. Exploit and Explore

• Exploit: To continue exploiting of the environment as it is in order to reap further benefits.

• Explore: To explore different states of the environment instead of keeping the same state too long.

2. Types of environments

• **Fully observable:** Agent knows **completely** about its current state.

Example: While running a car on the road (Environment), the driver (Agent) is able to see road conditions, signboard at a given time and drive accordingly. Road is threre for a fully observable.

• **Partially observable:** Agent know just **a part** of its current state.

Example: The vacuum cleaner (agent) only knows the dirt in the current room, but not the other rooms.

• **Unobservable:** Agent **don’t know** everything its current state.

• **Single agent:** Only one agent is responsible for all acts..

Example: An agent completing a crossword problem by itself.

• **Multiagent: Two** or **more** agents are taking actions in the environment.

Example: Chess is being played by an agent in a two agent environment.

• **Episodic:** Every state operates **independently** of the others. The next state is unaffected by the action in one state.

Example: A support bot (agent) will respond to one question, then another, and so on. Therefore, each question and answer is a separate episode.

• **Sequential:** The next state is **dependent** on the current action.

Example: Playing chess is a sequential action because one move affects the entire game.

• **Deterministic:** the next state is observable at a given time.

Example: The next light at a traffic signal is known to a pedestrian since it is a episodic environment.

• **Stochastic:** The next state is totally unpredictable for the agent.

Example: The listener is unaware of the next song because the radio station is a stochastic environment.

• **Other types:** Discrete vs continuous, Static vs dynamic, …

3. State representations

• **Atomic:** each state is indivisible, it has no internal structure.

• **Factored:** each state into a fixed set of variables or attributes, each of which can have a value.

• **Structured:** both see the details of the properties and see how they are related.

4. Solving Problems by Searching

A search problem is officially described as follows:

- **Initial state:** a starting state for the agent.

- **Possible actions:** actions that are applicable to the agent a given state.

    Notation: ACTIONS(state s) = { }

- **Transition model:** result of an action in a state.

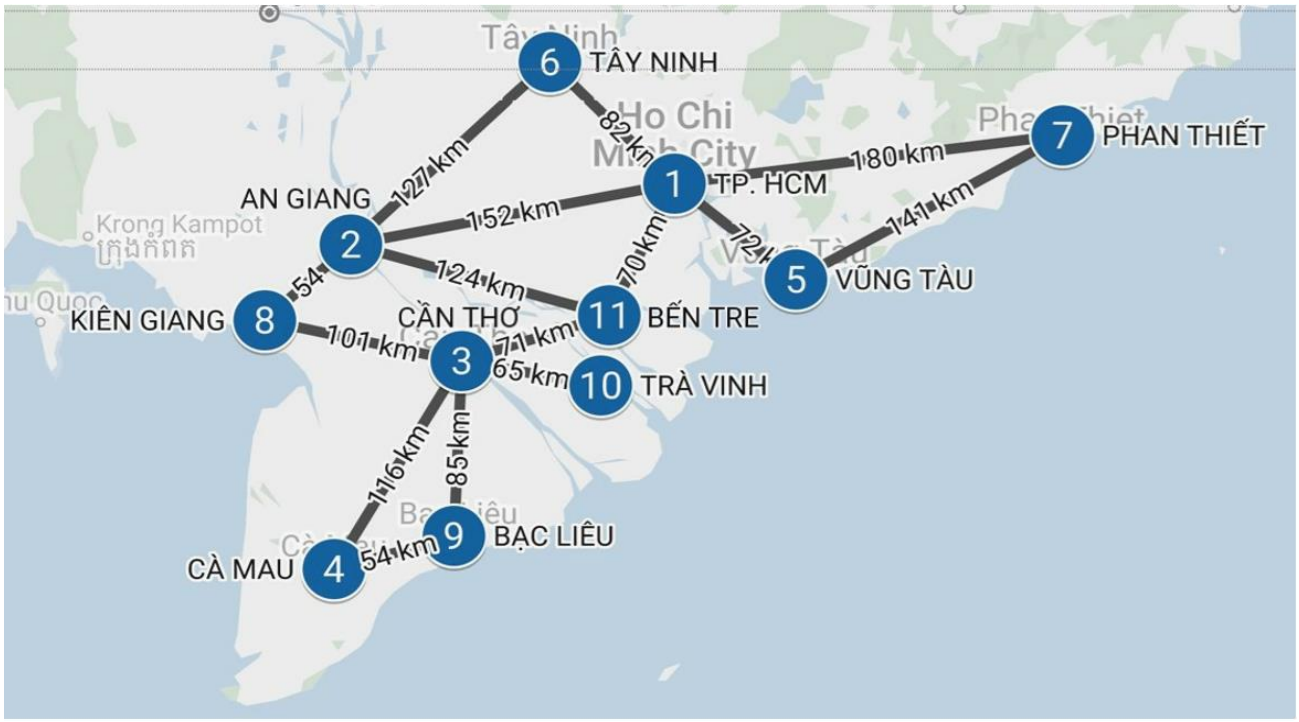
    Notation: RESULT(state s, acction a) = s’

- **Goal test:** checks to see if a state is a goal state.

- **Cost:**

    + **Step Cost:** Cost of performing an action.
    + **Path Cost**: Cost of performing all actions from start to finish.

Example: Find your way from PHAN THIET to AN GIANG.



- Initial state: PHAN THIET

- Possible actions: ACTIONS (PHAN THIET) = { go(VUNG TAU), go(TP.HCM) }

- Transition model:

RESULT (PHAN THIET, go(VUNG TAU)) = VUNG TAU

RESULT (PHAN THIET, go (TP.HCM)) = TP. HCM

RESULT (VUNG TAU, go(TP.HCM)) = TPHCM

RESULT (TP.HCM, go(TAY NINH)) = TAY NINH

RESULT (TP.HCM, go(BEN TRE)) = BEN TRE

RESULT (TP.HCM, go(AN GIANG)) = AN GIANG

RESULT (TAY NINH, go(AN GIANG)) = AN GIANG

RESULT (BEN TRE, go(AN GIANG)) = AN GIANG

- Goal test:

GOAL_TEST(TP.HCM) = False

GOAL_TEST(AN GIANG) = True

- **Cost:** best_path_cost(PHAN THIET, go(AN GIANG)) = 332 (PHAN THIET → TP.HCM → AN GIANG)

5. Searching for solutions

• Two main steps of search algorithms:

+ Choose a node.

+ Expanding the node.

• Two types of search algorithms:

+ Uninformed search.

+ Informed search.

6. Uninformed search algorithms

Uninformed search is a category of all-purpose search algorithms that uses a brute force approach to finding results. It is also known as blind search since uninformed search algorithms only know how to traverse the tree and have no other knowledge of the state or search space.

6.1. Breadth-first search (BFS)

6.1.1. Definition

BFS in which the root node is expanded first, then all the successors of the root node are expanded next, then their successors, and so on. Thus, even on infinite state spaces, this methodical search strategy is exhaustive.

6.1.2. Breadth-first search algorithm

```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
Step 1  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
Step 2  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
Step 3  frontier ← a FIFO queue with node as the only element
        explored ← an empty set
        loop do
Step 4      if EMPTY?(frontier) then return failure
Step 5      node ← POP(frontier)  /* chooses the shallowest node in frontier */
            add node.STATE to explored
Step 6      for each action in problem.ACTIONS(node.STATE) do
Step 7          child ← CHILD-NODE(problem, node, action)
                if child.STATE is not in explored or frontier then
Step 8              if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
Step 9              frontier ← INSERT(child, frontier)
```

Breadth-first search algorithm, return solution or failure (no result found).

• Step 1: Create a node with Sate = initial_sate(first state), path_cost = 0 (unexecuted).

• Step 2: Check if node is goal_sate, if yes then return solution.

• Step 3: Assign frontier = FIFO queue (fisrt in first out), containing only initial_sate.

• Step 4: If frontier is empty, return failure, otherwise go to step 5.

• Step 5: Take an element in the frontier and add it to the explored (possible actions).

• Step 6: Loop to all possible actions.

• Step 7: Return child_node (created from parent node when performing action).

• Step 8: If child is goal_test then return solution.

• Step 9: If child.sate is not in the explored or frontier set, add child to the frontier, go back to step 4.

6.1.3. Measuring algorithm's performance

• **Completeness:** the algorithm will find the solution of the problem if that solution exists.

• **Optimality:** the breadth-first search algorithm will find the most optimal solution provided that the path cost is a nondecreasing function of the depth.

• **Complexity:** the complexity of the breadth-first search algorithm is extremely large.

6.2. Uniform-Cost Search

6.2.1. Defititon

A path from the source to the destination is found using the least expensive cumulative cost through a search method known as uniform-cost search. Starting at the root, nodes are expanded in accordance with the minimum cumulative cost. A Priority Queue is then used to implement the uniform-cost search.

6.2.2. Uniform-Cost search algorithm

```
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure
Step 1  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
Step 2  frontier ← a priority queue ordered by PATH-COST, with node as the only element
        explored ← an empty set
        loop do
Step 3      if EMPTY?(frontier) then return failure
Step 4      node ← POP(frontier)  /* chooses the lowest-cost node in frontier */
Step 5      if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
            add node.STATE to explored
Step 6      for each action in problem.ACTIONS(node.STATE) do
Step 7          child ← CHILD-NODE(problem, node, action)
                if child.STATE is not in explored or frontier then
Step 8              frontier ← INSERT(child, frontier)
Step 9          else if child.STATE is in frontier with higher PATH-COST then
                    replace that frontier node with child
```

Uniform-Cost search algorithm, return solution or failure (no result found).

• Step 1: Create a node with Sate = initial_sate(first state), path_cost = 0 (unexecuted).

• Step 2: Assign frontier = priority queue (by path_cost), containing only initial_sate.

• Step 3: If frontier is empty, return failure, otherwise go to step 4.

• Step 4: Take an element in the frontier.

• Step 5: Check if node is goal_sate, if yes then return solution, if no add it to the explored (possible actions).

• Step 6: Loop to all possible actions.

• Step 7: Return child_node (created from parent node when performing action).

• Step 8: If child.sate is not in the explored or frontier, add child to the frontier.

• Step 9: If child.sate is in the frontier with a higher path_cost, replace child.sate with a node with a smaller path_cost.

6.2.3. Measuring algorithm's performance

• **Completeness:** the algorithm will find the solution of the problem if that solution exists.

• **Optimality:** the uniform-Cost search algorithm will find the most optimal solution provided.

• **Complexity:** the complexity of the uniform-Cost search algorithm is extremely large.

6.3. Depth-First Search (DFS)

6.3.1. Definition

The deepest node in the frontier is always expanded first in a depth-first search. It could be implemented as a call to BEST-FIRST-SEARCH where the evaluation function is the negative of the depth. It is typically implemented as a tree-like search rather than a graph search, which does not maintain a table of achieved states.

6.3.2. Depth-First Search algorithm

```
function Depth -FIRST-SEARCH(problem) returns a solution, or failure
Step 1  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
Step 2  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
Step 3  frontier ← a LIFO queue with node as the only element
        explored ← an empty set
        loop do
Step 4      if EMPTY?(frontier) then return failure
Step 5      node ← POP(frontier)  /* chooses the shallowest node in frontier */
            add node.STATE to explored
Step 6      for each action in problem.ACTIONS(node.STATE) do
Step 7          child ← CHILD-NODE(problem, node, action)
                if child.STATE is not in explored or frontier then
Step 8              if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
Step 9              frontier ← INSERT(child, frontier)
```

Depth-first search algorithm, return solution or failure (no result found).

• Step 1: Create a node with Sate = initial_sate(first state), path_cost = 0 (unexecuted).

• Step 2: Check if node is goal_sate, if yes then return solution.

• Step 3: Assign frontier = LIFO queue (last in first out), containing only initial_sate.

• Step 4: If frontier is empty, return failure, otherwise go to step 5.

• Step 5: Take an element in the frontier and add it to the explored (possible actions).

• Step 6: Loop to all possible actions.

• Step 7: Return child_node (created from parent node when performing action).

• Step 8: If child is goal_test then return solution.

• Step 9: If child.sate is not in the explored or frontier set, add child to the frontier, go back to step 4.

6.3.3. Measuring algorithm's performance

• **Completeness:** the depth-first search algorithm does not complete when there is an infinitely deep branch but no goal_state.

• **Optimality:** the depth-first search algorithm is nonoptimal.

• **Complexity**: the depth-first search algorithm is better than the breadth-first search algorithm in space.

6.4. Depth-Limited Search

6.4.1. Definition

We can use depth-limited search, a variation of depth-first search in which we give a depth limit and treat all nodes at depth as if they had no successors, to prevent depth-first search from wandering down an infinite route.

6.4.2. Depth-Limited Search

```
function DEPTH-LIMITED-SEARCH(problem, limit) returns a solution, or failure/cutoff
  return RECURSIVE-DLS(MAKE-NODE(problem.INITIAL-STATE), problem, limit)

Step 1  function RECURSIVE-DLS(node, problem, limit) returns a solution, or failure/cutoff
Step 2    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
Step 3    else if limit = 0 then return cutoff
          else
Step 4        cutoff_occurred? ← false
Step 5        for each action in problem.ACTIONS(node.STATE) do
                  child ← CHILD-NODE(problem, node, action)
                  result ← RECURSIVE-DLS(child, problem, limit − 1)
Step 6            if result = cutoff then cutoff_occurred? ← true
                  else if result ≠ failure then return result
Step 7        if cutoff_occurred? then return cutoff else return failure
```

Recursive_DLS, return solution or failure (no result found), cutoff (haven't found a solution yet).

• Step 1: Specify a limit parameter.

• Step 2: Check if node is goal_state, if yes return solution and finish.

• Step 3: If limit = 0, return cutoff.

• Step 4: Otherwise, cuttoff_occurred = false.

• Step 5: Perform a loop with each action, then possible actions will create child, limit - 1 (Recursive).

• Step 6: If result = cutoff then cuttoff_occurred = true, If result # failure return result.

• Step 7: If cutoff_occrred = True, return cutoff, otherwise return failure.

Depth-Limited Search algorithm , returns the Recursive_DLS function (recursive, limit)

6.4.3. Measuring algorithm's performance

• **Completeness:** the depth-Limited search algorithm does not complete.

• **Optimality:** the depth-Limited Search algorithm is nonoptimal.

• **Complexity:** the depth- Limited search algorithm is better than the depth-first search algorithm in space and time.

6.5. Iterative Deepening Search

6.5.1. Definition

An Iterative Deepening searching method that utilizes significantly less memory with each iteration (similar to Depth-First Search). IDS enforces a depth-limit on DFS to prevent becoming stranded in an infinite or extremely lengthy branch, hence achieving the necessary completeness. It does a left-to-right search on each branch of a node until it reaches the necessary depth. IDS then returns to the root node and investigates a different branch that is comparable to DFS.

6.5.2. Iterative Deepening Search algorithm

function ITERATIVE-DEEPENING-SEARCH(*problem*) **returns** a solution, or failure
  for *depth* = 0 **to** ∞ **do**
    result $\leftarrow$ DEPTH-LIMITED-SEARCH(*problem*, *depth*)
    if *result* $\neq$ cutoff **then return** *result*

Iterative Deepening Search algorithm, implement depth-limited Search algorithm, if result # cutoff, return result.

6.5.3. Measuring algorithm's performance

• **Completeness:** the Iterative Deepening Search algorithm does complete.

• **Optimality:** the Iterative Deepening Search algorithm will find the most optimal solution provided that the path cost is a nondecreasing function of the depth.

• **Complexity:** the Iterative Deepening Search algorithm is the best in time and space.

6.6. Comparing uninformed search strategies

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening
Complete?	Yes[a]	Yes[a,b]	No	No	Yes[a]
Time	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$
Space	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$
Optimal?	Yes[c]	Yes	No	No	Yes[c]

Evaluation of tree-search strategies. b is the branching factor; d is the depth of the shallowest solution; m is the maximum depth of the search tree; l is the depth limit. Superscript caveats are as follows: [a] complete if b is finite; [b] complete if step costs $\geq \epsilon$ for positive ϵ; [c] optimal if step costs are all identical; [d] if both directions use breadth-first search.

7. Informed search algorithms

Unlike Uninformed Search which has to look through search space for all possible solutions to the problem without having any additional knowledge about search space. Informed search is better and more useful since it is provided evaluation functions. This will help them have more knowledge about how far we are from the goal, path cost and how to reach to goal node or how far each way is to your destination.

7.1. Evaluation functions f(n)

The main difference between uninformed and informed search is Evaluation functions. It makes informed search more useful for large search space. Evaluation function f(n) is a cost estimate from initial to goal through node n. There are different ways to estimate distance like estimating by the bird's flight route.

For example: estimating distance from HCM to An Giang. Assuming the bird's flight route from HCM City to An Giang: 150km. We can use this flight path value as f (n)

7.2. Idea:

A node will be chosen for expansion based on an evaluation function f(n) and a node with f(n) smaller will be chosen in the priority queue for expanding

- **Algorithms:**

imformation search

```
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  frontier ← a priority queue ordered by PATH-COST, with node as the only element
  explored ← an empty set                                   f(n)
  loop do
    if EMPTY?(frontier) then return failure                          (1)
    node ← POP(frontier)  /* chooses the lowest-cost node in frontier */   (2)
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)      (3)
    add node.STATE to explored
    for each action in problem.ACTIONS(node.STATE) do
      child ← CHILD-NODE(problem, node, action)                      (4)
      if child.STATE is not in explored or frontier then
        frontier ← INSERT(child, frontier)                           (5)
      else if child.STATE is in frontier with higher PATH-COST then
        replace that frontier node with child          f(n)          (6)
```

This informed search is almost like uniform-cost search; the only difference is path-cost, which is changed to evaluation function f(n) in informed search.

At the beginning, we will define initial node with initial state and goal state, then we define a frontier with a priority queue ordered by f(n), and initial node is the first element, and the last one we define is an empty explored for if node is expanded or not.

In loop, at step (1), we will check if frontier is an empty or not then return failure or select the lowest-cost node in frontier to expand in step (2).

At step (3), after choosing the lowest-cost node, we will check if that node is a goal or not. If not we will add it to an explored or return a solution if it is true.

At step (4), we will go through each action that node can possible do. For each action of node, we will get its child by the function Child-Node. Then we will check that child if it isn't expanded or not already in the frontier. And insert it into the frontier if a condition is True, at step (5).

Then at step (6), we will compare f(n) between a child which is already in frontier with a child and replace that frontier node with a lower f(n) child

Type of informed search

Depending on different choices of f(n), we will have different informed search algorithms:

7.3. Best-first search (bfs)

- **Define:**

  Best-first search is an example of the general Tree-Search or Graph-Search algorithm in which a node is chosen for expansion based on an evaluation function f(n). Furthermore, a node which is choose for expansion is the lowest evaluation. And the evaluation function that is used in best-first search is heuristic function, denoted h(n), then f(n)=h(n)

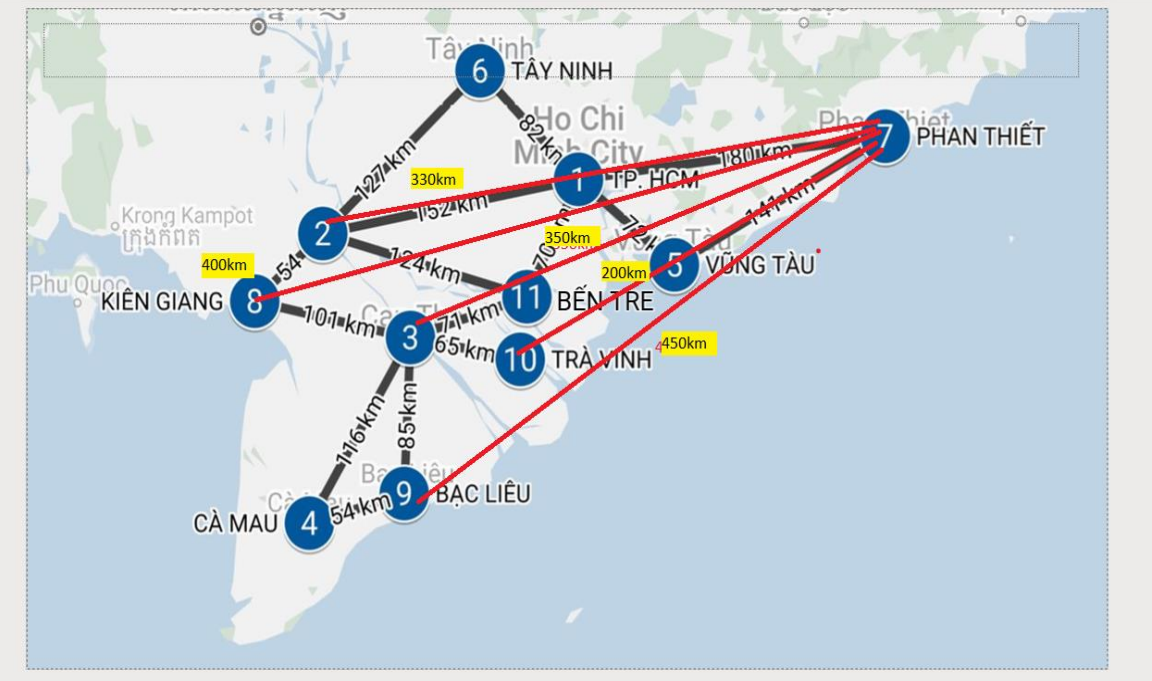
- **Heuristic function, h(n):**

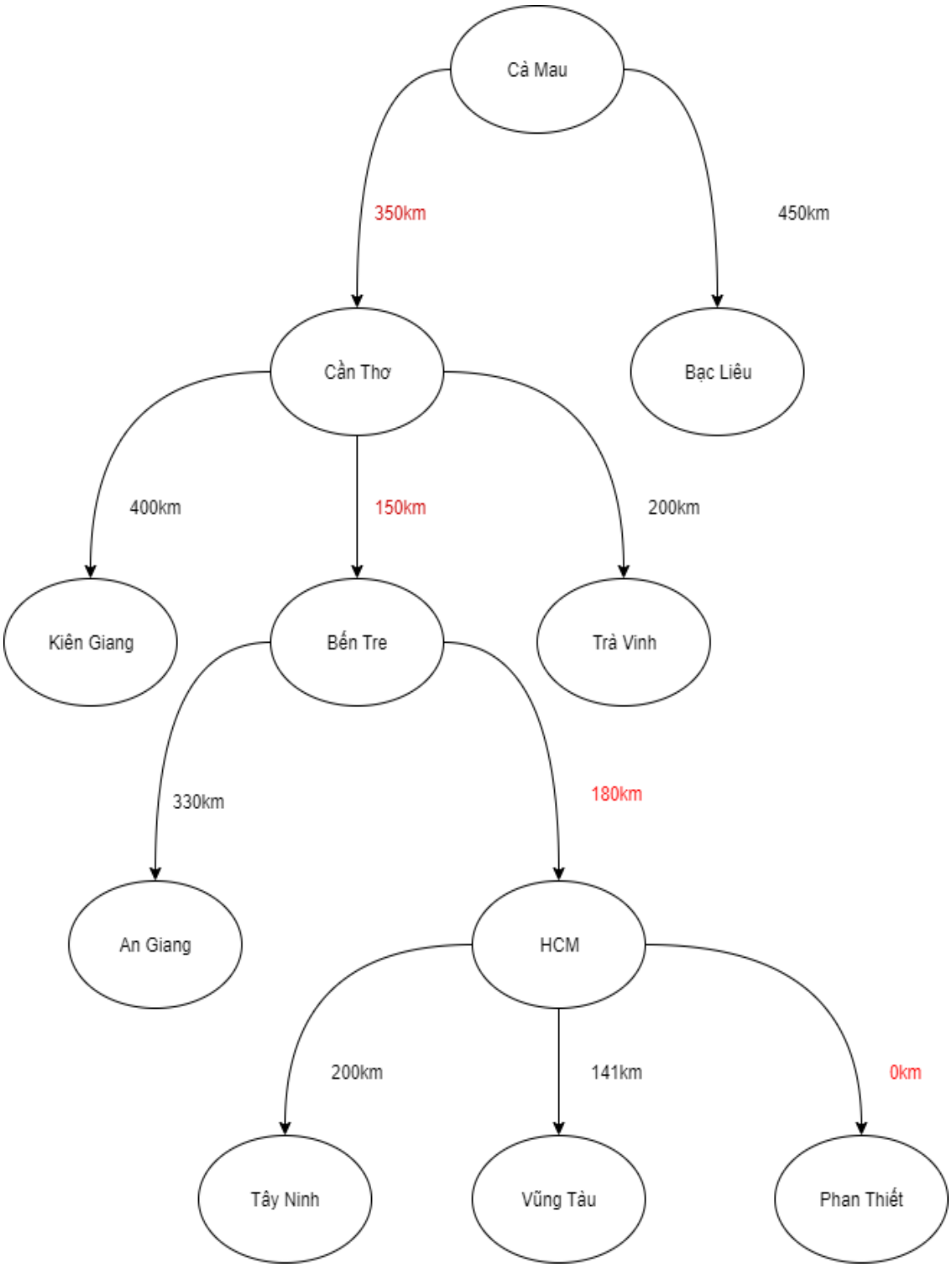
  It is the estimated cheapest cost from the node n to goal state. If n is a goal node, then h(n)=0.

- **Demo:**

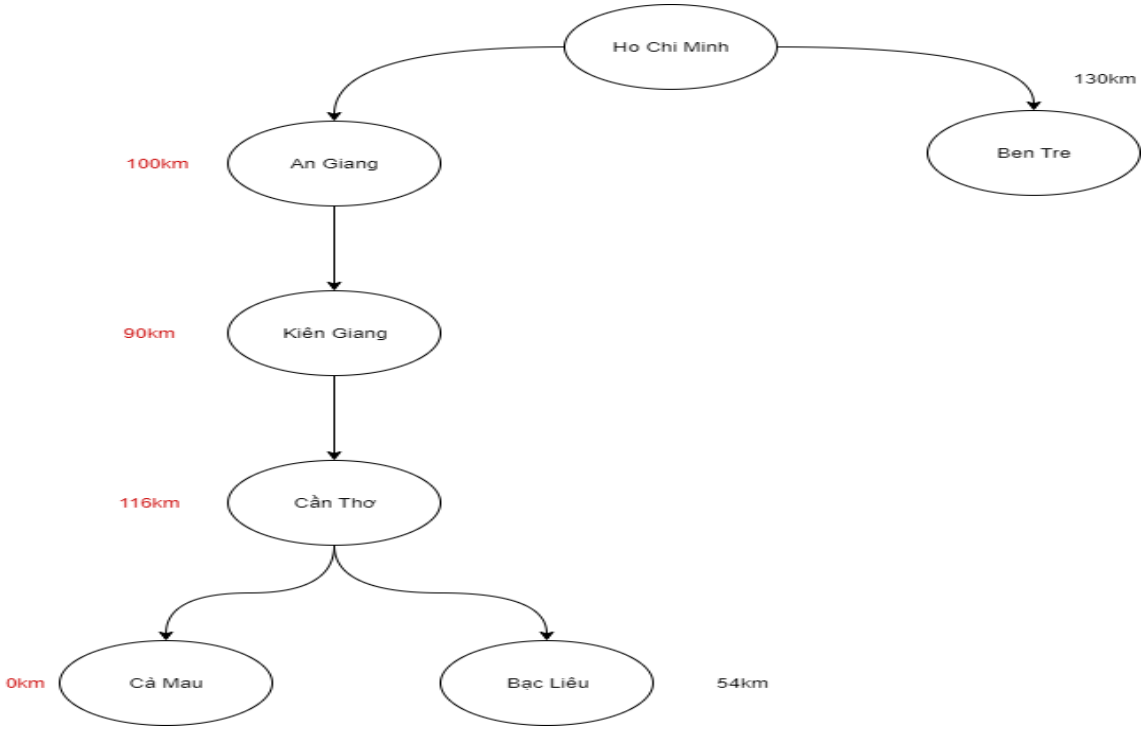
  Find the best route to go to Phan Thiet from Ca Mau.

  The distance calculated by the flight path from node n to the destination.





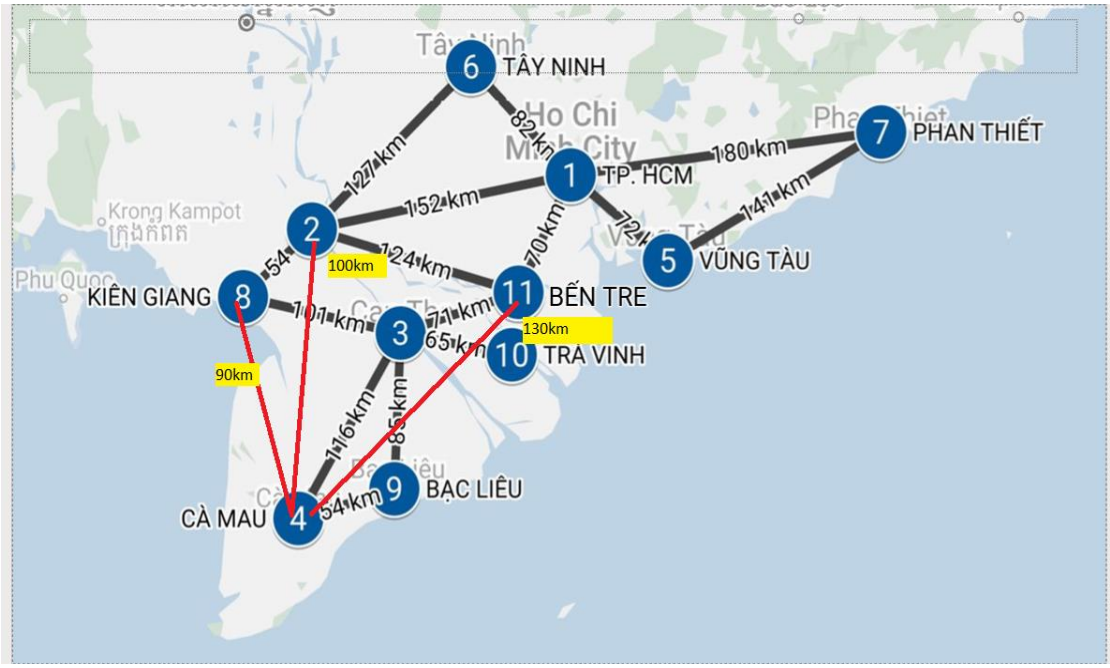
In this demo, Ca Mau is an initial. From Ca Mau we can expand two different nodes such as Can Tho and Bac Lieu. Base on f(n) we can see Can Tho with f(n) is smaller than Bac Lieu then we choose Can Tho and expand it. Expanding Can tho leads to many



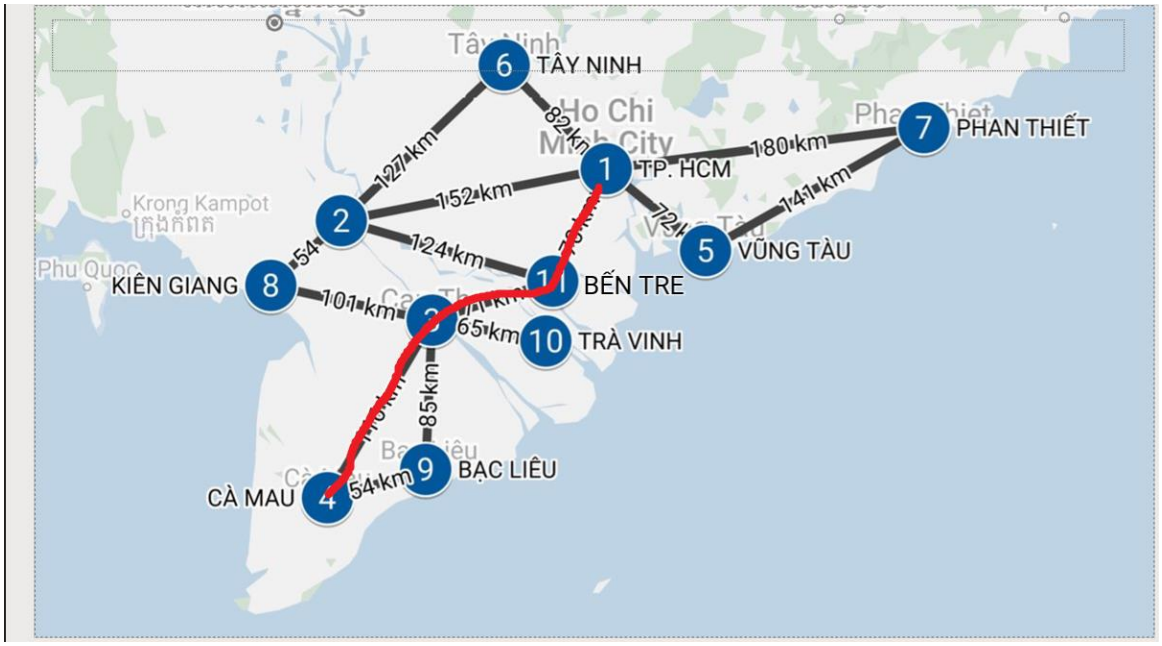
different nodes such as Kien Giang, Ben Tre, Tra Vinh. Depending on f(n) Ben tre is a node selected to expand, we have Tay Ninh, Vung Tau, Phan Thiet. And Phan Thiet is the goal of the problem.

- **Best first search is not an optimal Algorithms:**

  For example: Find the shortest route to go Ca Mau from Ho Chi Minh



As we can see, based on heuristic function the shortest way to go to Ca Mau from Ho Chi Minh city is Ho Chi Minh to An Giang to Kien Giang To Can Tho and last point is Ca Mau. But according to the map, if we calculate the real path cost base on map we can get 423km for the route that is found by best first search. And the real path cost for the route from Ho Chi Minh , Ben Tre, Can Tho and Ca Mau is just 257km. It is smaller than 423km, then the shortest route must be Ho Chi Minh, Ben Tre, Can Tho and Ca Mau.



Reasons why best-first search is not optimal algorithms:

- Do not have a path-cost
- Heuristic is not always estimate true

7.4. A-Star Search

Definition:

A-Star Search is an better algorithms than best-first search algorithms since it fix all drawbacks of best-first algorithms by upgrading the evaluate function. It will evaluate nodes by combing g(n), the cost to reach the node, and h(n), the cost to get from the node n to the goal. Then the evaluation function of A-star will be:

F(n)= g(n)+h(n)

G(n): The path cost from the initial node to node n

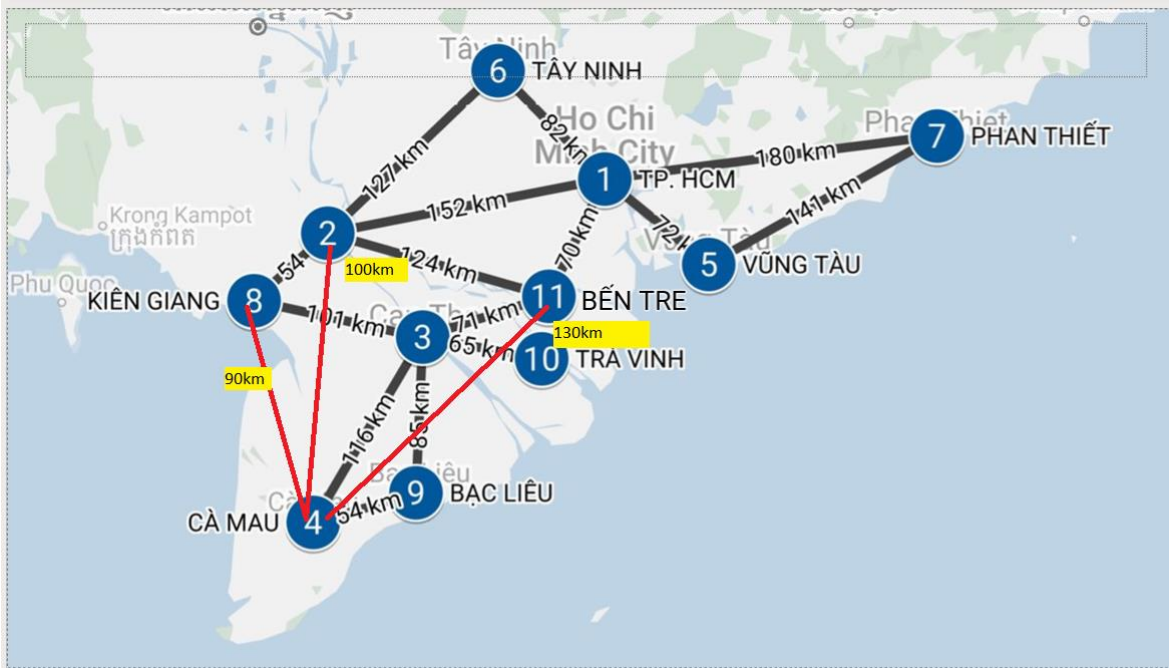
H(n): The cheapest path from node n to the goal

F(n): cost of cheapest solution through n

Demo

For example: Finding a shortest route to go to Ca Mau from Ho Chi Minh

First, we still estimate distance from node n to goal by the bird's flight route.



Then we will have:

In the next step, this will be different from best-first search algorithm when the evaluation f(n)=g(n)+h(n). As we can see, expanding Ho Chi Minh node leads to An Giang node and Ben Tre node. With An Giang, its heuristic h(n) is 100km and path-cost is 152km then its f(n) will be 252km. Doing the same thing, we will have f(n) of An Giang is 200km then this node is selected for expansion. In the following step we can realize that f(n) of Can tho is bigger than An Giang (257>252) then we select An Giang node to expand and do the same things for the rest of nodes. Finally, the shortest route that we found by A-star is Ho Chi Minh, Ben Tre, Can Tho and Ca Mau.

So we can see that A-star algorithm is better than Best-First search algorithms because of g(n). But it is still not enough for optimal.

- **Conditions for optimal.**

We can remember that two reasons why best-first search is not optimal are path-cost and a bad heuristic. Therefore, the condition we require for optimality is that h(n) be a consistent heuristic.

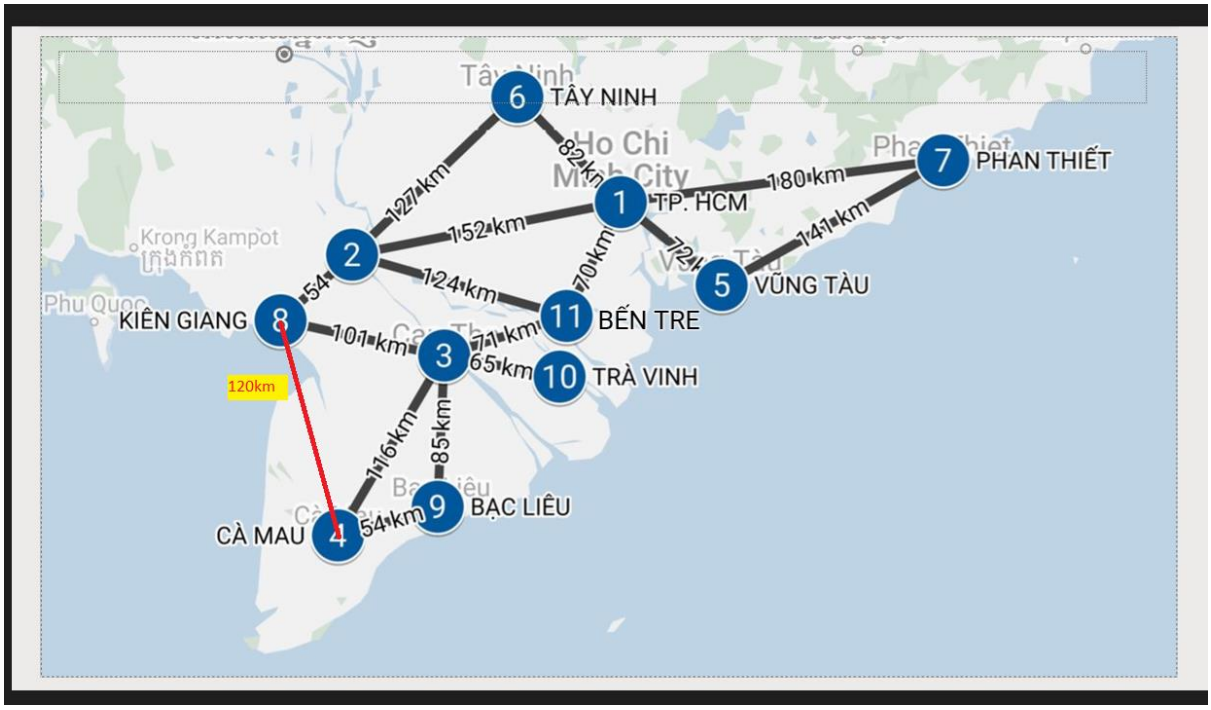
- **Consistent heuristic:**

This is similar to triangle inequality, a heuristic h(n) is consistent if, for every node n and every successor n' of n generated by any action a, the estimated cost of reaching the goal from n is no greater than the step cost of getting to n' plus the estimated cost of reaching the goal from n':

H(n) <= cost(n,n') +h(n')

- **Example of consistent heuristic:**

Consider node n in Kien Giang



H(Kien Giang)=120km

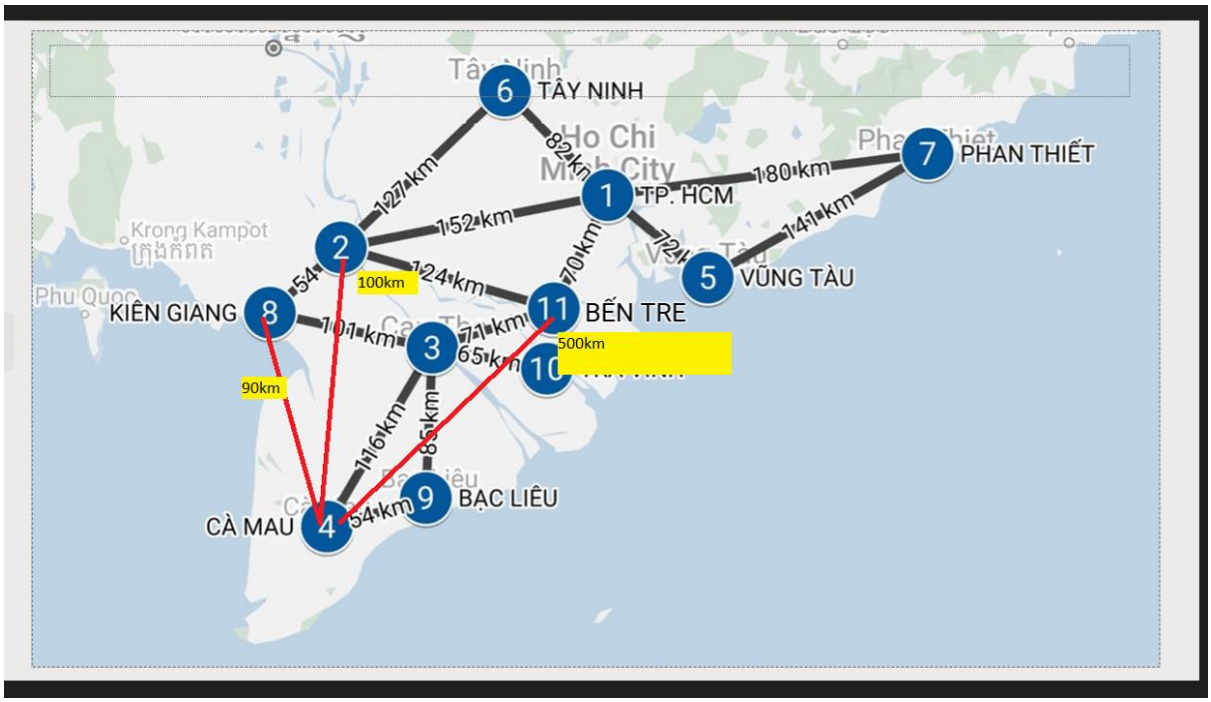
Cost(Kien Giang, Can Tho)=101km

H(Can Tho)=116km

H(Kien Giang)<=cost(Kien Giang, Can Tho)+h(Can tho)(120<=101+116=217)

- **Example: Inconsistent heuristic**

  Demon again instance, finding the best route to go Ca Mau from Ho Chi Minh with an inconsistent heuristic.



H(Ben Tre)=500km

Cost(Ben Tre, Can Tho)=71km

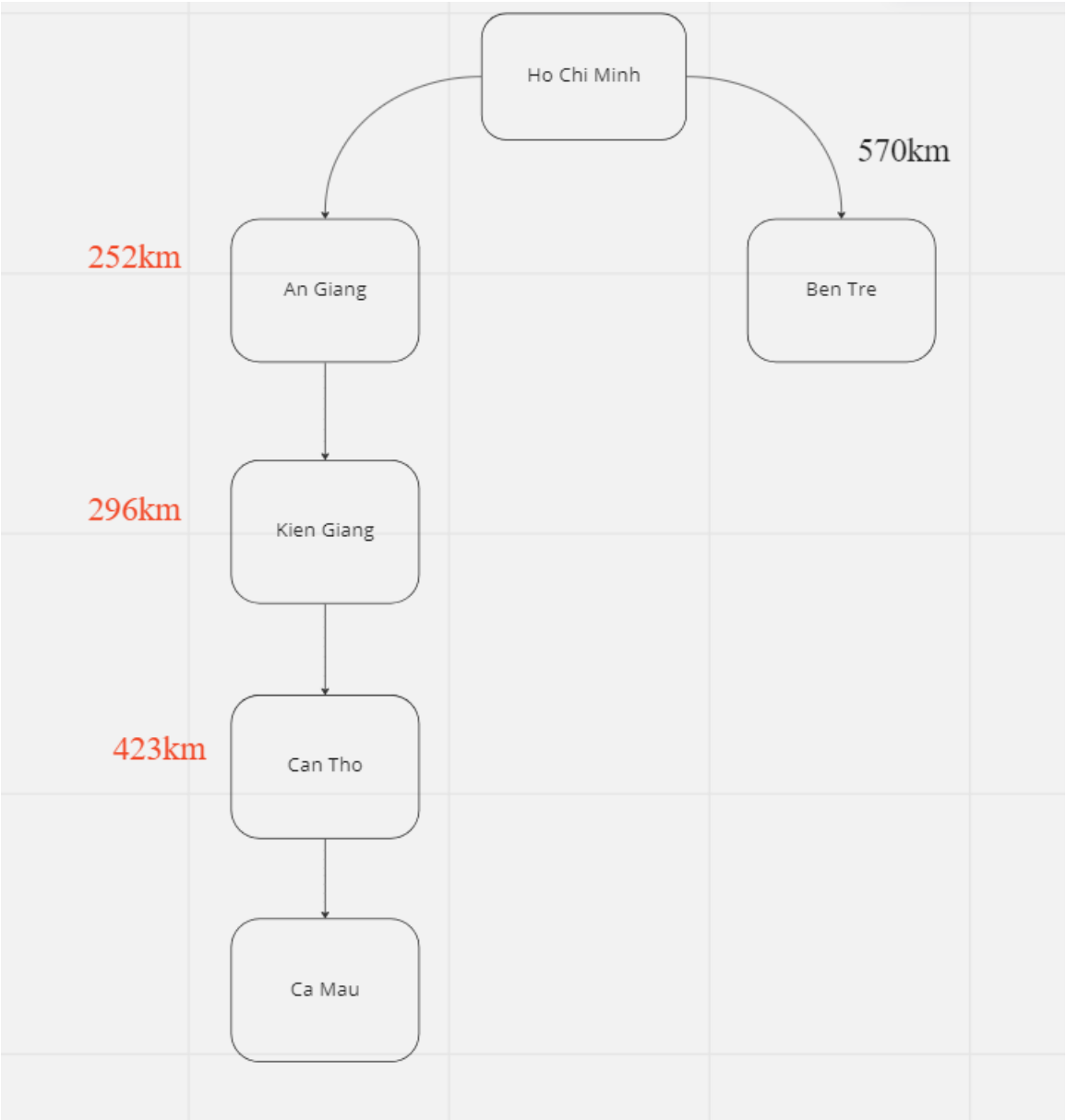
H(Can Tho)=116km

H(Ben Tre)<= cost(Ben Tre, Can Tho) +h(Can Tho)

(500> 71+116=187)

⇨ This heuristic is inconsistent

With inconsistent heuristics, the problem will be solved in another way



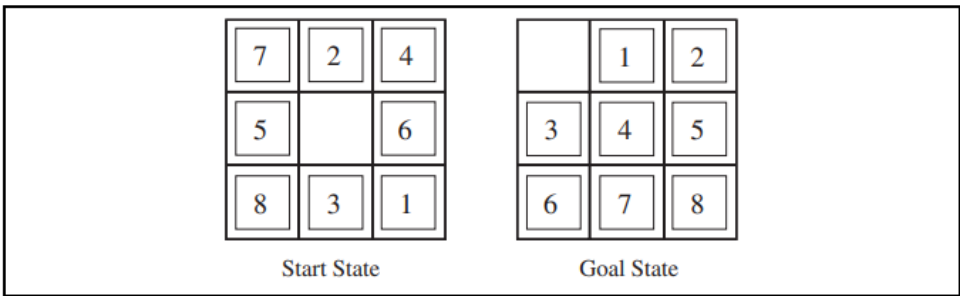
We do not go through Ben Tre anymore since the f(n) of Ben Tre is bigger than An Giang (570 >252). And when we do step by step we can have another route (Ho Chi Minh, An Giang, Kien Giang, Can Tho, Ca Mau) which is not the shortest route. Therefore, if the heuristic is not consistent the problem may be solved in the wrong ways.

7.5. Heuristic functions comparison

As we know with many different heuristics, we will have many different informed search algorithms. And the accuracy and efficiency of informed search algorithms are based on the heuristic. Therefore, the bigger heuristic is, the faster algorithm is. If the heuristic is consistent, the maximum value is equal actual cost.

For example:

Consider the heuristic for the 8-puzzle



H1= the number of misplaced tiles

H2=the sum of the distances of the tiles from their goal positions.

Note: tiles cannot move along diagonals

According to the description of h1, h2, we have:

H1=8

H2=3+1+2+2+2+3+3+2=18

	Search Cost (nodes generated)			Effective Branching Factor		
d	IDS	A*(h_1)	A*(h_2)	IDS	A*(h_1)	A*(h_2)
2	10	6	6	2.45	1.79	1.79
4	112	13	12	2.87	1.48	1.45
6	680	20	18	2.73	1.34	1.30
8	6384	39	25	2.80	1.33	1.24
10	47127	93	39	2.79	1.38	1.22
12	3644035	227	73	2.78	1.42	1.24
14	–	539	113	–	1.44	1.23
16	–	1301	211	–	1.45	1.25
18	–	3056	363	–	1.46	1.26
20	–	7276	676	–	1.47	1.27
22	–	18094	1219	–	1.48	1.28
24	–	39135	1641	–	1.48	1.26

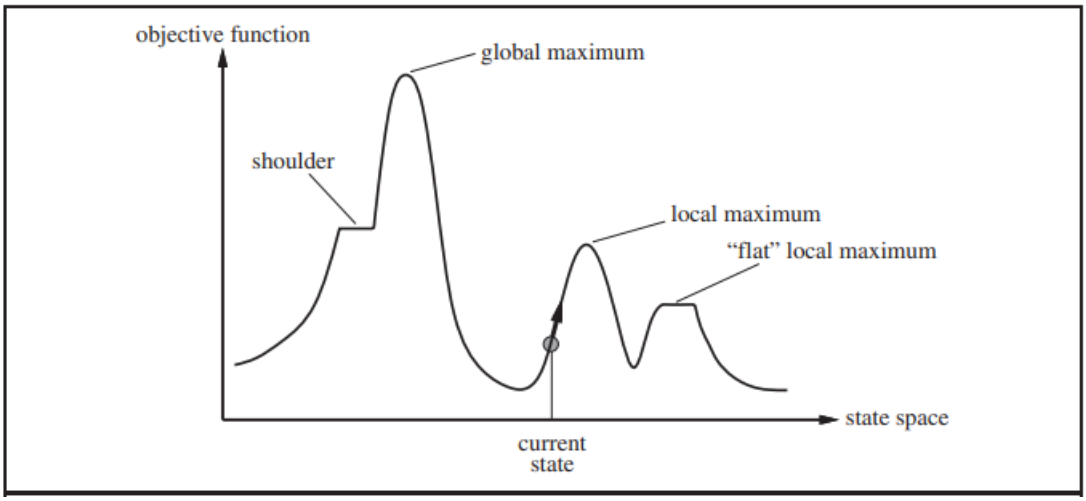
As we can see, if the heuristic is bigger, the algorithms will be more effective and use less memory.

V. LOCAL SEARCH – SEARCHING FOR GOAL STATE.

1. Compare Local Search, Uninformed Search and Informed Search

	Informed Search	**Uninformed Search**	**Local Search**
Known as	It is also known as Heuristic Search	It is also known as Blind Search	It is also knows as heuristic method
Performance	Finding a solution more quickly	Finding solution slow as compared to an informed search	Finding a solution maximizing a criterion among a number of candidate solutions
Completion	May or may not be complete	Always complete	Incomplete
Cost Factor	Cost is low	Cost is high	Cost is low
Time	Consuming less time	Consuming moderate time	Consuming moderate time
Examples of Algorithms	A* Search Best-First Search	Depth First Search Breadth First Search	Hill-Climbing search

2. State-space landscape



The state-space landscape is a very useful way to understand local search. A landscape has both “Location” (defined by the state) and “elevation” (defined by the value of the heuristic cost function or objective function). It will help to find the global maximum, local maximum, “flat” local maximum, and shoulder.

A complete local search algorithm always finds a goal if one exists and an optimal algorithm always finds a global minimum/maximum.

3. Local Search

Definition:

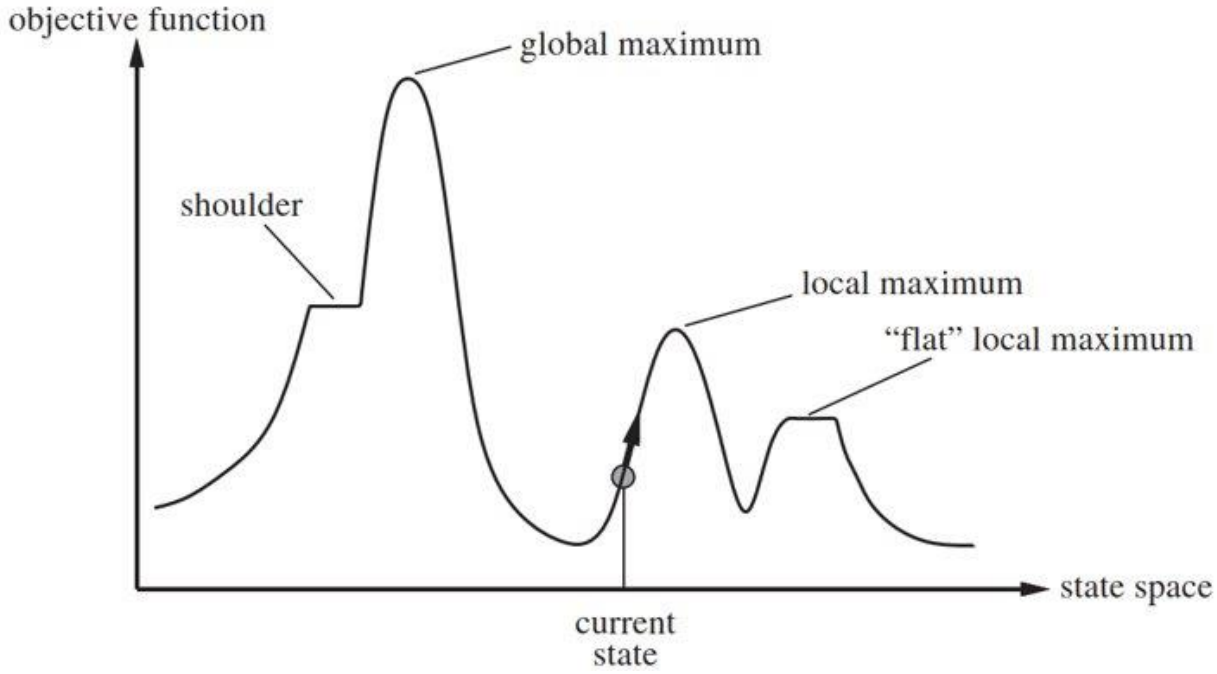
Local search is a heuristic method for searching in more complex environments. This algorithm will help people find the best goal state based on what objective function that we use instead of the route to the goal state

Idea:

Local search algorithms start from a randomly chosen complete instantiation and move from one complete instantiation to the next. And they just store only the current node and its successors whenever it runs then it uses very little memory, usually a constant amount.

4. Hill-climbing search algorithm

4.1. Description



The hill-climbing search is an algorithm belonging to the local search group which are used in complex Environments . It keeps track of one current state and on each iteration moves to the neighboring state with highest value from that neighbors, it heads in the direction that provides the steepest ascent. When it reaches a "peak" where no neighbor's value is higher, it ends. Hill climbing does not look ahead beyond the immediate neighbors of the current state. This is like trying to find Mount Everest's summit in a dense fog when amnesic.

4.2. Simple Hill Climbing Algorithm

Step 1 : **function** HILL-CLIMBING(*problem*) **returns** a state that is a local maximum

Step 2 : *current* ← MAKE-NODE(*problem*.INITIAL-STATE)

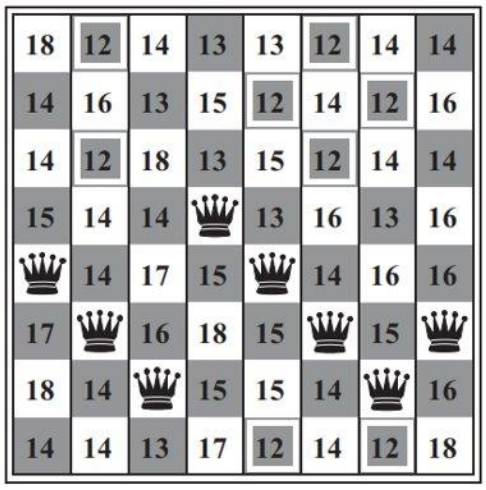
Step 3 + 4 : **loop do**

Step 5 : *neighbor* ← a highest-valued successor of *current*
if neighbor.VALUE ≤ current.VALUE **then return** *current*.STATE → Step 6
current ← *neighbor*

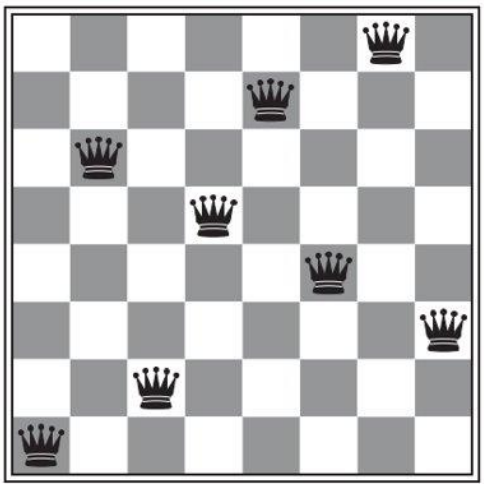
- **Step 1:** If the starting state is the goal state, evaluate it, return success, and stop.
- **Step 2:** Initiate a random node as current state
- **Step 3:** Loop until a solution is discovered or every operator has been used.
- **Step 4:** Choose an operator and use it on the current state.
- **Step 5:** Check the new state:
    1. If the new state is better to the current state, the old state shall be designated as the current state.
    2. Check whether it is goal state or not, if true , return success and quit.
    3. Else if not, then return to step 3.
- **Step 6:** Exit.

4.4. Examples

To illustrate hill climbing, we will use the 8-queens problem .We will use a complete-state formulation, which means that every state has all the components of a solution, but they might not all be in the right place. In this case every state has 8 queens on the board, one per column. The initial state is chosen at random, and the successors of a state are all possible states generated by moving a single queen to another square in the same column (so each state has 8 x 7 = 56 successors). The heuristic cost function is the number of pairs of queens that are attacking each other; this will be zero only for solutions.(It counts as an attack if two pieces are in the same line, even if there is an intervening piece between them.) Picture 1 shows a state that has h = 17. It also shows the h values of all its successors.



Picture 1



Picture 2

5. Issues of Hill-climbing search

5.1. Too much successors

5.1.1. Problem

Because there are too many successors, it will be difficult or long to consider which neighbors have the largest value because each time to consider which successor has the highest value, it has to go through all the successors in the successor's array, so it will waste time as well as the waste of resources. For example, if our agent are walking on a hill, it may have to traverse an array of different 360 degrees to calculate which degrees will have the highest value, then the function must check through 360 successors.

5.1.2. Solution

Stochastic Hill Climbing which picks one of the uphill moves at random; the probability of selection varies with the uphill move's steepness. Even while this typically converges more slowly than the steepest climb, it may discover superior solutions in certain state landscapes.

Stochastic hill climbing is implemented via first-choice hill climbing, which generates successors at random until one is produced that is superior to the existing state. This is an effective tactic when a state has numerous (thousands) successors.

5.2. Local optimum

5.2.1. Problem

A hill-climbing algorithm that never makes “downhill” moves toward states with lower value(or higher cost) but there might be another state also present which is higher than the local maximum

5.2.2. Solution

Method 1 :Random-Restart Hill climbing which basically is a series random restarts of Hill climbing Search, which follows the maxim "If at first you don't succeed, try, try again." .It conducts a series of hill-climbing searches from randomly generated initial states, until a goal is found.

Method 2 : Local beam search - Instead of running random restarts initial state from the beginning and waiting for the state to reach its destination, local beam search runs multiple initial states at once. It is just like multiple threads running concurrently and interacting with each other, and selecting threads that contain the successors with the highest value. However, there are still some problem that there are too many successors . Stochastic beam search is a solution for that problem which is combination of stochastic Hill climbing and local beam search, instead of finding the successors with the largest value, they only choose threads that contain the successors with better value.

6. Simulated Annealing

6.1. Description

Simulated annealing is such an algorithm. In metallurgy, annealing is the process used to temper or harden metals and glass by heating them to a high temperature and then gradually cooling them, thus allowing the material to reach a low-energy crystalline state.

6.2. Simulated Annealing Algorithm

Mechanism: The simulated-annealing algorithm has a general design that resembles hill climbing. But it chooses a random move rather than the best one. It is always accepted if the action makes the situation better. In the absence of this, the algorithm accepts the move with a probability that is less than 1. The probability decreases exponentially with the “badness” of the move—the amount ΔE by which the evaluation is worsened. The probability also decreases as the “temperature” T goes down: “bad” moves are more likely to be allowed at the start when T is high, and they become more unlikely as T decreases. If the *schedule* lowers T to 0 slowly

enough, then a property of the Boltzmann distribution, $e^{\Delta E/T}$, is that all the probability is concentrated on the global maxima, which the algorithm will find with probability approaching 1.

Pseudo code :

```
Step 1 :    function SIMULATED-ANNEALING(problem, schedule) returns a solution state
              inputs: problem, a problem
                      schedule, a mapping from time to "temperature"
Step 2:     current ← MAKE-NODE(problem.INITIAL-STATE)
Step 3:     for t = 1 to ∞ do
Step 3.1:       T ← schedule(t)
Step 3.2:       if T = 0 then return current ──────────► Step 4
Step 3.3:       next ← a randomly selected successor of current
Step 3.4:       ΔE ← next.VALUE – current.VALUE
                if ΔE > 0 then current ← next
                else current ← next only with probability e^(ΔE/T)
```

***Input :** a problem and the "temperature" T.

***Output :** a solution state .

- **Step 1:** If the starting state is the goal state, evaluate it, return success, and stop.
- **Step 2:** Initiate a random node as current state
- **Step 3:** Loop the below operation until find out the solution state

    1. Initial the "temperature" T for the move with a probability

    2. Check whether the temperature equal to 0 or not , if true then return the current state ,that means a solution is discovered.

    3.If not , choosing a random successors in the successors array

    4.Then check whether the action makes the situation better or not . If true , If the new state is better to the current state, the old state shall be designated as the current state.

- **Step 4:** Exit.

7. Searching in NONDETERMINISTIC environments

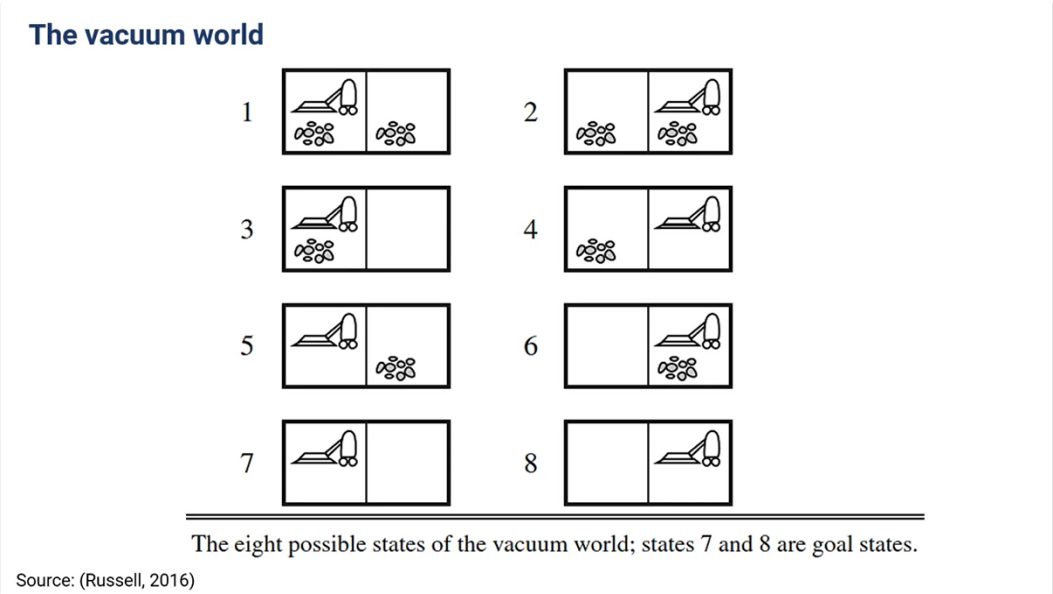
7.1. Searching in NONDETERMINISTIC environments

7.1.1. Description

When the environment is nondeterministic, the agent does not know the state it will transition to after taking an action. This indicates that an agent will now think, "I'm either in state s1 or s3, and if I do action an I'll end up in state s2, s4, or s5" instead of, "I'm in state s1 and if I do action an I'll end up in state s2."

In nondeterministic environments, the solution to a problem is no longer a sequence, but rather a conditional plan (sometimes called a contingency plan or a strategy) that specifies what to do depending on what percepts agent receives while executing the plan. We examine nondeterminism in this section and partial observability in the next.

7.1.2. For Example



The eight possible states of the vacuum world; states 7 and 8 are goal states.

Source: (Russell, 2016)

We have 2 changes:

- **Transition model**: RESULTS function

  RESULTS(1, Suck)= {5,7}

- **Solution**:

  [Suck,if State = 5 Then [Right,Suck] else[]].

7.2. And_Or Graph Search

7.2.1 .Component of a AND-OR tree

The AND / OR tree consists of OR and AND nodes:

- **OR node:** (state) The only branching is introduced by the agent's own choices in each state: I can do this action or that action
- **AND nodes**: (links -> successors) Must handle all actions to come up with different solutions for each problem.

7.2.2 .Solution in AND-OR tree

Solution for an AND-OR tree is a subtree that:

- A goal node at every leaf
- OR node: specifies one action at each
- AND node: includes every outcome branch at each

7.2.3 .AND-OR-SEARCH Algorithm

```
Step 1:   function AND-OR-GRAPH-SEARCH(problem) returns a conditional plan, or failure
            OR-SEARCH(problem.INITIAL-STATE, problem, [])
Step 2:
          ------------------------------------------------------------------------
Step 2.1: function OR-SEARCH(state, problem, path) returns a conditional plan, or failure
Step 2.1.1:   if problem.GOAL-TEST(state) then return the empty plan
Step 2.1.2:   if state is on path then return failure
Step 2.1.3:   for each action in problem.ACTIONS(state) do
                plan ← AND-SEARCH(RESULTS(state, action), problem, [state | path])
                if plan ≠ failure then return [action | plan]
              return failure
          ------------------------------------------------------------------------
Step 2.2: function AND-SEARCH(states, problem, path) returns a conditional plan, or failure
Step 2.2.1:   for each s_i in states do
                plan_i ← OR-SEARCH(s_i, problem, path)
Step 2.2.2:   if plan_i = failure then return failure
              return [if s_1 then plan_1 else if s_2 then plan_2 else ... if s_{n-1} then plan_{n-1} else plan_n]
```

Figure 4.11 An algorithm for searching AND–OR graphs generated by nondeterministic environments. It returns a conditional plan that reaches a goal state in all circumstances. (The notation [x | l] refers to the list formed by adding object x to the front of list l.)

***Input :** a problem.

***Output :** a conditional plan , or failure(This doesn't mean that there is no solution from the current state; it simply means that if there is a noncyclic solution, it must be reachable from the earlier incarnation of the current state, so the new incarnation can be discarded).

Step 1 : Initiate an OR Node as the Initial state

Step 2 : Using recursive to find out the solution plan by operating in harmony and support each other between OR-SEARCH Function and AND-SEARCH function which are showed below

2.1. OR-SEARCH

- **Step 1:** Check whether the starting state is the goal state or not, evaluate it. If true, return a empty plan, and stop.
- **Step 2: If not ,**Check whether the process is in a loop by checking the current state is in the path or not. If true, return failure, and stop.
- **Step 3:** Else, Operating all the action , And enter states generated by this action into the AND-SEARCH function to expand the path , and loop until find out the solution plan or get failure

2.2. AND-SEARCH

- **Step 1 :** Enter states generated by this action into the OR-SEARCH function to expand the path , and loop until find out the solution plan or get failure
- **Step 2:** Check whether the process can find out the solution or not , If true , return a solution plan to get to goal state from the initial state.

Step 3 : Exit.

8. Searching in Partially Observable Environments

• The agent's percepts do not suf-fice to pin down the exact state.

• Unknown or observed loss of part of the environment.

• Advantages:

+ Saving on the price of buying and installing sensors.

+ Give sensors more time to collect data.

• Disadvantages:

+ Can't handle many problems.

+ Ineffective troubleshooting.

• Belief states: the set of states an agent is capable of assuming.

8.1. No observable – current state in unknown

Ideal:

- Build belief state

Initial state: all possible states in reality

possible actions: ACTIONS(belief state b) → b = {s1, s2}

actionsp(s1) = {a, b, c, e}

actionsp(s2) = {a, c, d}

we can choose actions(c) = {a, b, c, d, e} or actions(b)= {a, b}

Transition model: RESULT (belief state b, action a) = new belief state b'

b = {s1, s2}

b' = Resultp(s1, a) U Resultp (s2, a)

Goal test: all states in belief state to be goal state

b = {s1, s2}

if goal-testp(s1) and goal-testp(s2) both return true, then goal-test(b) is true

Cost: step-cost(b) || path-cost(b)

- Convert the problem to fully observable.
- Use algorithms BFS, DFS, A*, AND-OR, ... to solve.
- **Note**: If the physical problem has **N state**, then you have 2^N **belief states.**

8.2. Partially observable

Similar to no observable, however the transition model will be more focused by inserting the belief state into the perception, determining whether the condition is met, and then either sending the state back into the belief or removing it from the belief state depending on the result.

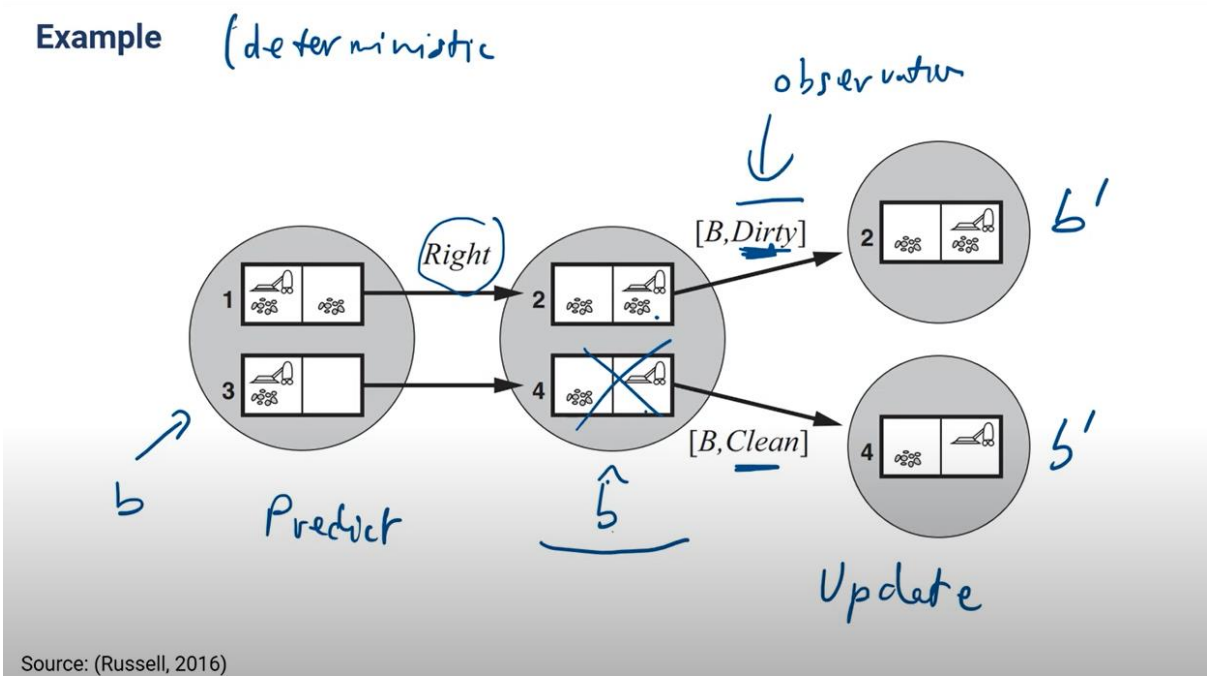
State prediction: RESULT(b, a) state action, prediction.

Observation: O(o: o = PERCEPT (s) and s belong to b) defining the set of acceptances that can be obstructed.

State update: determines, for each possible perception, the percept will result from the perception.

b'= {s: o = PERCEPT(s) and s belong to b

Example:



Source: (Russell, 2016)

Following to the figure we can see the chaining that: observation -> update -> narrow the results.

8.3. Demonstration for algorithms

Example: Localization robot

Given a **map**.

Robot has only **4 sonar sensors**:

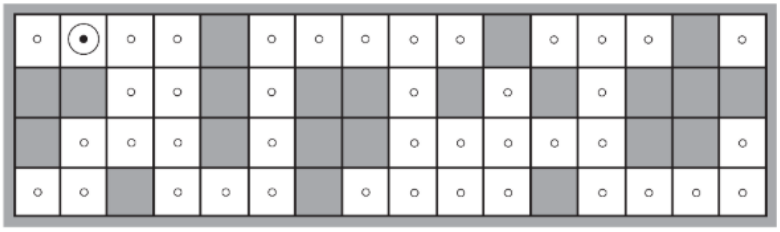
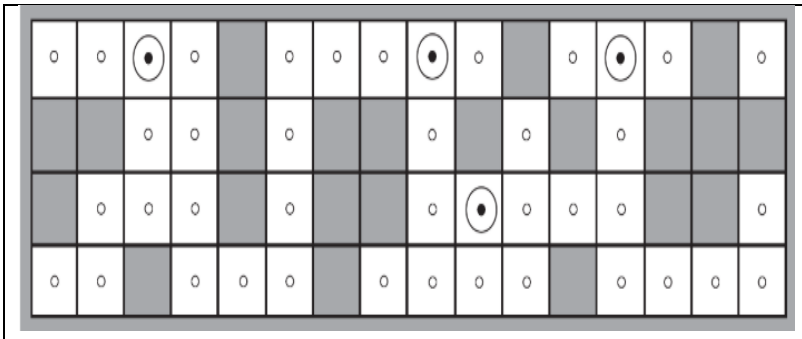
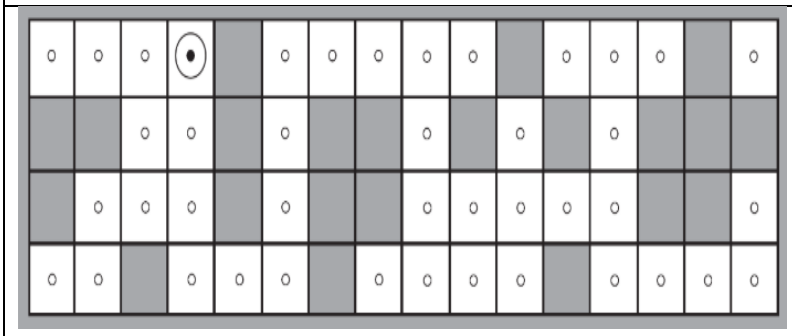


Image: (Russell, 2016)

	Step 1. The agent moves to the right The initial Belief state is all positions. Percept has a barrier above it → Belief state remaining 4 states.
	Step 2. Continues to move right Percept has a wall on the top and right side. → Belief state is left with one state.

9. Online Search

9.1. Definition of online search

Online search is a real-world algorithm that also learns how to solve problems without a transition model. The more often you search online, the more intelligent it will become.

Online search is a good idea in dynamic or semi dynamic domains.

For instance: The weather forecast, robot build map in a building, etc.

Online Search algorithms is useful in

- There is a penalty for (too long computation).
- Non-deterministics environments
- No environment model.

9.2. Compare online and offline search:

	Online search	**Offline search**
Element	Initial state Possible action, Goal test, Cost | Path cost. But no transition model due to the inability to perceive the surroundings. Use explorer to find the solution.	Initial state Possible action Transition model Goal test Cost | Path Cost.
Cost	Return competitive radio	Return best cost
Result	Penalties Return only one result like (actions)	For agents that are stationary or repeating a state, there is no workaround. Await the end result of the entire execution.
Suitable level	Unknown environment has too many results but returns only one that optimizes search time and volume.	Can be deterministic or non-deterministic problems. Focus only on results or all results.
Action	Actions -> return a list of actions allowed in state	Action -> return an new action

Environment model	NO	Yes
Example for application	Programming intelligent robots to automatically deliver goods, there are many ways to deliver goods, but agents only care about the best and most optimal way. If the robot takes too long to deliver, causing the receiver to wait or lose the order, the rating point will be forfeited. Agents just need to know where the destination of the order is and how to manage them to overcome the problems.	Program an AI to show the shortest path between 2 cities, just give it enough maps and weights between the city distances, the result it returns is always the shortest path from between those 2 cities.

9.3. Competitive ratio:

In online search, there are two types of path costs:

- Actual path cost: the actual cost that the agent takes, including steps to explore the environment and find a solution.
- Shortest path cost: the shortest cost to find the solution.

Competitive ratio = actual path cost / shortest path cost (algorithm efficiency)

Dead-end state

Some actions are irreversible, they load to a dead-end state:

For example: fail in the hole, 1-way street, endless loop

Safely explorable state space: don’t have dead-end state.

9.4. Online Depth-first search agent:

• Input: (s') percept indicates the current state of the environment.

• Output: for each state the agent thinks he is in, what action should it take, or in other words return an action for a state.

***Algorithm description**

```
1 function ONLINE-DFS-AGENT(s') returns an action
  2 inputs: s', a percept that identifies the current state
  3 persistent: result, a table indexed by state and action, initially empty
                untried, a table that lists, for each state, the actions not yet tried
                unbacktracked, a table that lists, for each state, the backtracks not yet tried
                s, a, the previous state and action, initially null

  4 if GOAL-TEST(s') then return stop
  5 if s' is a new state (not in untried) then untried[s'] ← ACTIONS(s')
  6 if s is not null then
    6.1 result[s, a] ← s'
    6.2 add s to the front of unbacktracked[s']
  7 if untried[s'] is empty then
    7.1 if unbacktracked[s'] is empty then return stop
    7.2 else a ← an action b such that result[s', b] = POP(unbacktracked[s'])
  8 else a ← POP(untried[s'])
  9 s ← s'
 10 return a
```

Figure 4.21 An online search agent that uses depth-first exploration. The agent is applicable only in state spaces in which every action can be "undone" by some other action.

Pseudo code of Online-DFS-Agent

[1] the input is the (s') state of the current environment, and for each state, what action should the agent take.

[2] inputs (s' : the current state the agent thinks it is in)

[3] persistent (global variable)

Result : returns a table containing (s, a , s') , which means the agent is standing at state s and performing action a gets state s'

Untried : an array of states and actions that I haven't tried yet

Unbacktracked : an array containing the states of the previous states that it has not considered.

[4] Check if (s') is the target state, if it is, stop.

[5] If (s') is a new state (not in untried), then we put Action(s') in untried[s'].

[6] If s is not null (it's just null at the start) then we pass [7].

> [6.1] We put [s, a] ← s' into the result table (that is, write a line in state s perform action a and get state s')
>
> [6.2] Add(s) and unbacktracked[s']

[7] If action of (s') is empty (no action to perform anymore), then we pass [8]

[7.1] If unbacktracked[s'] is empty (there is no state to return to) then we stop.

[7.2] Otherwise, unbacktracked[s'] is not an empty set, then we perform action b to go back to the previous state to perform (backtrack)

[8] If in [7], the action of (s') is not an empty set, then we get an action from untried[s']

[9] s' will now be s.

[10] return action a.

Properties of the algorithm:

• Completeness: the algorithm will find a solution if the problem has a solution

• The execution time of online search will be longer than offline search.

9.5. Online A* search (find a good solution quickly):

Idea: at each step, we choose a successor's according to the estimated cost (cost(s,s') + H(s'))

H(s) = h(s) (initially at will be the hueristic bird distance, then the agent will both walk in reality and update H)

9.6. Learning Real – Time A* agent

• Input: (s') percept indicates the current state of the environment.

• Output: for each state the agent thinks he is in, what action should it take, or in other words return an action for a state.

***Algorithm description**

```
1 function LRTA*-AGENT(s') returns an action
2   inputs: s', a percept that identifies the current state
3   persistent: result, a table, indexed by state and action, initially empty
                H, a table of cost estimates indexed by state, initially empty
                s, a, the previous state and action, initially null

4   if GOAL-TEST(s') then return stop
5   if s' is a new state (not in H) then H[s'] ← h(s')
6   if s is not null
6.1   result[s, a] ← s'
6.2   H[s] ← min_{b ∈ ACTIONS(s)} LRTA*-COST(s, b, result[s, b], H)
10  a ← an action b in ACTIONS(s') that minimizes LRTA*-COST(s', b, result[s', b], H)
11  s ← s'
12  return a

7 function LRTA*-COST(s, a, s', H) returns a cost estimate
8   if s' is undefined then return h( )
9   else return ( . . ') + [ ']
```

Figure 4.24 LRTA*-AGENT selects an action according to the values of neighboring states, which are updated as the agent moves about the state space.

Pseudo code of LRTA AGENTS(s')*

[1] The input is the current (s') state, the result will be an action.

[2] inputs (s', a)

[3] persistent (global variable)

Result: returns a table containing (s, a, s'), which means the agent is standing at state s and performing action a gets state s'

H: an estimator of the path from state s to the destination, initialized to empty

[4] If (s') is the target state then we stop.

[5] If (s') is a new state, not in H, then we initialize H[s^']←h(s^')

[6] If s is not null (only starting state s will be assigned null)

[6.1] We put [s, a] ← s' into the result table (that is, write a line in state s perform action a and get state s')

[6.2] Update H[s] with min value in LRTA*- COST(s, b, result[s, b], H) (b belongs to Action(s)), state s has many actions, each action has a different cost, we choose the action with the lowest cost.

[7] The input to the LRTA*-COST is, some state s, performing action a, obtaining the unknown state s'.

[8] If s' is not already in H, initialize h(s).

[9] Otherwise, if it is already in H, we update c(s, a, s') + H[s']

[10] Choose an action that gives the lowest cost to the next move

[11] s' becomes s.

[12] Returns action a.

Properties of the algorithm:

- Completeness: the algorithm will find a solution if the problem has a solution
- Improved execution time because it uses estimated cost

10. Constraint Satisfaction Problems

Previous algorithms use atomic state representation (Black box). Now we move to factored representation.

10.1. Constraint satisfaction problems (CSP)

Atomic state representations are states that are comparable to entities that we cannot see or fully define.

• Factored representation: A state consists of a number of variables

Find the assignment to state so that all values satisfy the requirement (this is a challenge using the structure of factored representations).

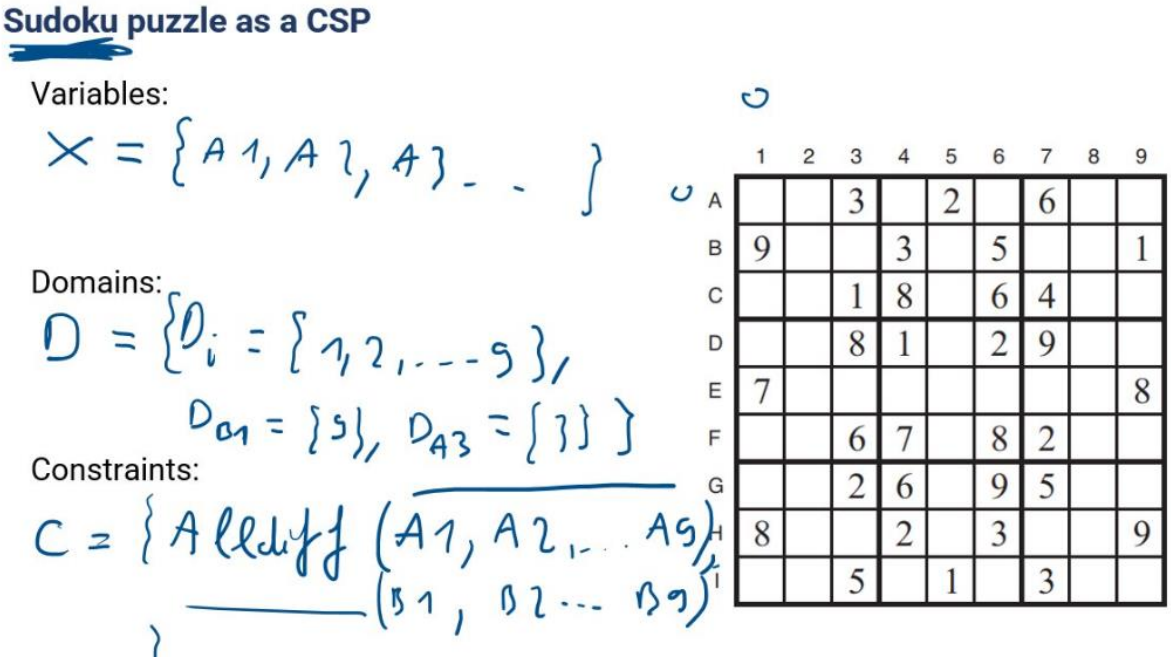
• CSP consists of three elements:

X (set of all variables) (set of all variables)

D (the range of possible values for variables)

C (collection of constraints that variables must satisfy) (set of constraints that variables must satisfy)

• To solve CSP, assign values to all variables that fall inside its domains in order to satisfy all constraints.



The example in the class

10.2. Constraint graph

• Nodes: are all variables.

• Links: are 2 variables in any binding.

10.3. Constraint propagation

Idea: narrow the domains of the surrounding variables by using variables whose domains are 1 value. (csp solves problems faster than normal search algorithms such as (BFS, DFS, A*,...), in many cases where the search space is too large, the search algorithms can't find it but CSP can't find it. can still be found.)

• Specifically: Use constraints to limit the scope of variables; when a variable's scope is reduced, the scope of its second and third siblings may also be affected (propagation). We have discovered the answer by limiting the range of each variable to only one value.

10.4. Local consistency (node, arc, path)

• Consistent: Variables are referred to as consistent when they comply with limitations.

• Local consistency: This is the ability of a collection of variables that are close to one another to satisfy the requirements.

• To maintain local consistency, I make sure that variables that are close to one another fulfill all conditions before propagating that condition satisfaction to nearby variables. (The variable becomes inconsistent when domain values are removed.)

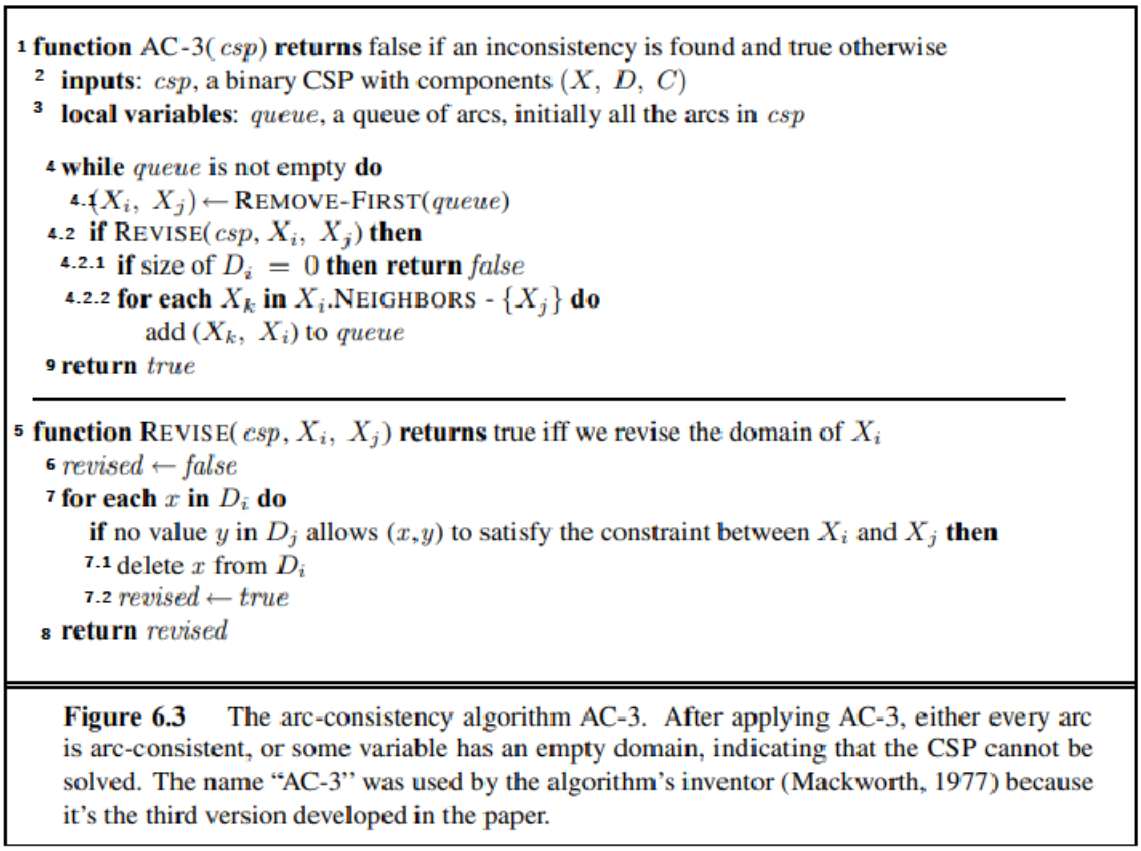
10.5. AC3 algorithm (arc consistency)

• Input: a binary CSP of a problem with components (X, D, C)

• Output: + False if the constraint is not satisfied

+ True if the constraint is satisfied

• **Algorithm explanation**



```
1 function AC-3( csp) returns false if an inconsistency is found and true otherwise
2   inputs: csp, a binary CSP with components (X, D, C)
3   local variables: queue, a queue of arcs, initially all the arcs in csp

4   while queue is not empty do
4.1   (X_i, X_j) ← REMOVE-FIRST(queue)
4.2   if REVISE(csp, X_i, X_j) then
4.2.1   if size of D_i = 0 then return false
4.2.2   for each X_k in X_i.NEIGHBORS - {X_j} do
            add (X_k, X_i) to queue
9   return true

5 function REVISE( csp, X_i, X_j) returns true iff we revise the domain of X_i
6   revised ← false
7   for each x in D_i do
        if no value y in D_j allows (x,y) to satisfy the constraint between X_i and X_j then
7.1     delete x from D_i
7.2     revised ← true
8   return revised
```

Figure 6.3 The arc-consistency algorithm AC-3. After applying AC-3, either every arc is arc-consistent, or some variable has an empty domain, indicating that the CSP cannot be solved. The name "AC-3" was used by the algorithm's inventor (Mackworth, 1977) because it's the third version developed in the paper.

Pseudo code of AC3 algorithm

[1] Their AC-3 algorithm input is a csp consisting of 3 components (X, D, C)

[2] inputs (csp, a binary csp with components (X, D, C))

[3] queue (set of constraints with 2 variables)

[4] Run through all those constraints in turn

[4.1] Get a constraint of 2 variables (Xi, Xj), then put (Xi, Xj) into the Revise function

[4.2] If the Revise function returns true (with a narrowing of the domain of Xi), then we pass [4.2.1]

[4.2.1] If the domain set of Di = 0 (meaning no value), then we return false (problem has no solution)

[4.2.2] Select the variable Xk that is adjacent to the variable Xi. Add (Xk , Xi) to the queue

[5] The Revise function takes as input (csp, Xi, Xj) and returns true if it narrows the domain of Xi, false if it cannot narrow the domain of Xi.

[6] Set the revised variable to false

[7] Retrieve each x value in the domain set Di of Xi. Considering that there is no y value in the domain set Dj of Xj , to help (x, y) satisfy the constraint, we pass [7.1].

[7.1] Remove x from the domain set Di

[7.2] Assign revised to true

[8] Returns revised

[9] If the constraints in the queue have been reviewed, return true.

10.6. Path consistency

• Used for 3-variable constraints.

• We have a set of 2 variables { (Xi, Xj) } which is considered path – consistency , when we have { Xi = a, Xj = b} then we find the value for Xm to satisfy constraints with {Xi , Xm} and {Xm, Xj}. And if not found, there exists a pair of values (a, b) making it inconsistent.

10.7. K-consistency: (generalized form of local consistency)

Idea: get a set with K -1 variables, then find the value for the K-th variable, if found, it reaches K-consistency, if not found, it does not reach K-consistency.

10.8. Global constraints

• Definition: is a constraint with an arbitrary number of variables

• Example: Alldiff constraint, Atmost constraint.

o Alldiff(x1, x2, x3, x4)

o For the 9x9 sodoku problem:

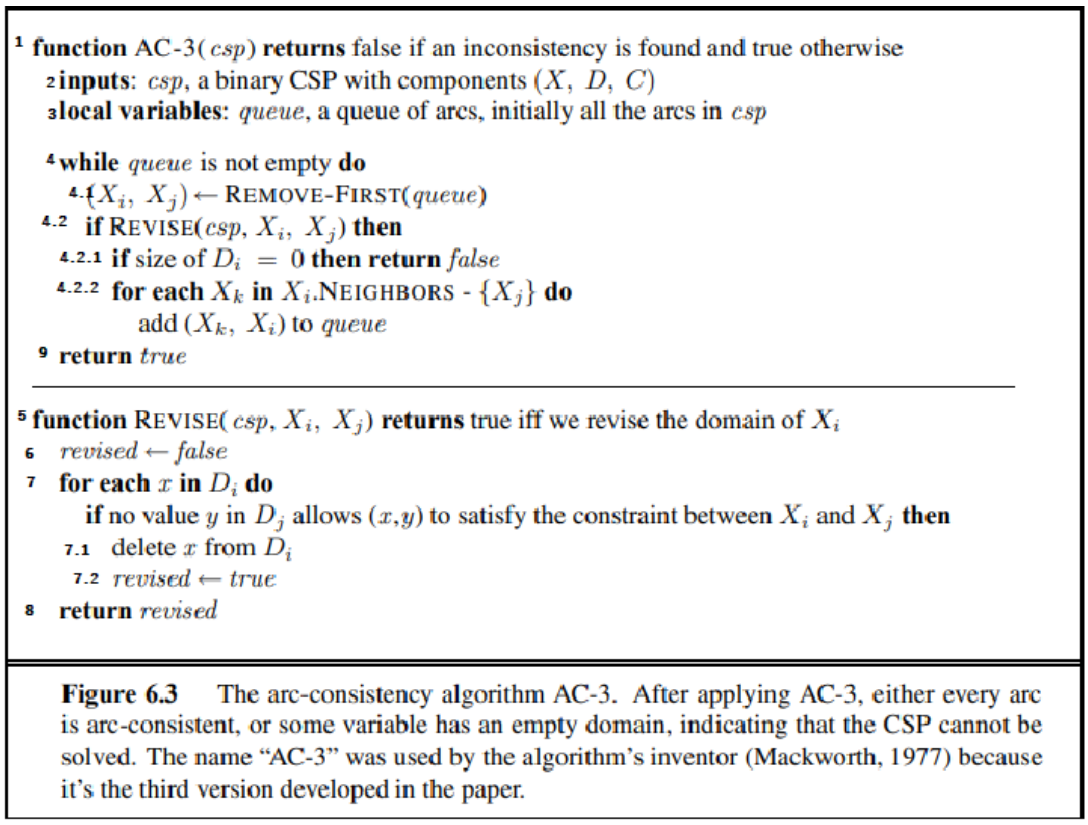
o Alldiff(0,1, 2, 3, 4, 5, 6, 7, 8)

10.9. Solving a Sodoku puzzle: (AC – 3, Triplets)

• Input: a binary csp of a problem with components (X, D, C)

• Output: + False if the constraint is not satisfied

+ True if the constraint is satisfied.



```
1 function AC-3(csp) returns false if an inconsistency is found and true otherwise
2 inputs: csp, a binary CSP with components (X, D, C)
3 local variables: queue, a queue of arcs, initially all the arcs in csp

4 while queue is not empty do
4.1   (Xi, Xj) ← REMOVE-FIRST(queue)
4.2   if REVISE(csp, Xi, Xj) then
4.2.1   if size of Di = 0 then return false
4.2.2   for each Xk in Xi.NEIGHBORS - {Xj} do
            add (Xk, Xi) to queue
9 return true

5 function REVISE(csp, Xi, Xj) returns true iff we revise the domain of Xi
6   revised ← false
7   for each x in Di do
        if no value y in Dj allows (x,y) to satisfy the constraint between Xi and Xj then
7.1     delete x from Di
7.2     revised ← true
8   return revised
```

Figure 6.3 The arc-consistency algorithm AC-3. After applying AC-3, either every arc is arc-consistent, or some variable has an empty domain, indicating that the CSP cannot be solved. The name "AC-3" was used by the algorithm's inventor (Mackworth, 1977) because it's the third version developed in the paper.

Pseudo code of AC-3 algorithm

[1] Their AC-3 algorithm input is a csp consisting of 3 components (X, D, C)

[2] inputs (csp, a binary csp with components (X, D, C))

[3] queue (set of constraints with 2 variables)

[4] Run through all those constraints in turn

[4.1] Get a constraint of 2 variables (Xi, Xj), then put (Xi, Xj) into the Revise function

[4.2] If the Revise function returns true (with a narrowing of the domain of Xi), then we pass [4.2.1]

[4.2.1] If the domain set of Di = 0 (meaning no value), then we return false (problem has no solution)

[4.2.2] Select the variable Xk that is adjacent to the variable Xi. Add (Xk , Xi) to the queue

[5] The Revise function takes as input (csp, Xi, Xj) and returns true if it narrows the domain of Xi, false if it cannot narrow the domain of Xi.

[6] Set the revised variable to false

[7] Retrieve each x value in the domain set Di of Xi. Considering that there is no y value in the domain set Dj of Xj , to help (x, y) satisfy the constraint, we pass [7.1].

[7.1] Remove x from the domain set Di

[7.2] Assign revised to true

[8] Returns revised

[9] If the constraints in the queue have been reviewed, return true.

10.10. Backtracking search algorithm

• Idea: If there are variables with domains larger than 1, we should pick a value to attempt, enter it, and then assign it to other cells. If we can assign it to all cells, we will have found the variable. the value we initially chose was incorrect, so we delete it from the domain of that cell. However, if, after a number of steps, we come across a cell whose domain is empty, we are unable to identify the solution.

• Main steps:

+ Step 1: select unassigned variables, try assigning a value to it

+ Step 2: check that value if it is inconsistent with the current assignment then we try another value, if it is consistent with the current assignment then we continue back to step1.

• Pros of algorithm: Improved over DFS, since backtracking uses the commutative property.

• Input : csp

• Output : solution or failure

• Algorithm description:

```
1 function BACKTRACKING-SEARCH(csp) returns a solution, or failure
2   return BACKTRACK({ }, csp)

3 function BACKTRACK(assignment, csp) returns a solution, or failure
4   if assignment is complete then return assignment
5   var ← SELECT-UNASSIGNED-VARIABLE(csp)
6   for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
6.1   if value is consistent with assignment then
6.1.1     add {var = value} to assignment
6.1.2     inferences ← INFERENCE(csp, var, value)
6.1.3     if inferences ≠ failure then
6.1.3.1     add inferences to assignment
6.1.3.2     result ← BACKTRACK(assignment, csp)
6.1.3.3     if result ≠ failure then
              return result
6.2   remove {var = value} and inferences from assignment
7   return failure
```

Figure 6.5 A simple backtracking algorithm for constraint satisfaction problems. The algorithm is modeled on the recursive depth-first search of Chapter 3. By varying the functions SELECT-UNASSIGNED-VARIABLE and ORDER-DOMAIN-VALUES, we can implement the general-purpose heuristics discussed in the text. The function INFERENCE can optionally be used to impose arc-, path-, or k-consistency, as desired. If a value choice leads to failure (noticed either by INFERENCE or by BACKTRACK), then value assignments (including those made by INFERENCE) are removed from the current assignment and a new value is tried.

Backtracking-Search algorithm pseudo code

[1] Input is a csp consisting of components (X, D, C) and returns a solution (if all variables have been assigned values) or failure

[2] return BackTrack ({}, csp)

[3] BackTrack function accepts assignment, csp (for sodoku problem, the initial assignment is already valid cells), (if there is nothing, put the empty set in the assignment)

[4] If the assignment is complete, which means all variables are assigned a unique value, then the assignment is also the solution, we return the assignment.

[5] If the assignment is incomplete, we select a variable that has not been assigned a value.

[6] Assign the selected variable a value from its domain set.

[6.1] If the value just assigned to that variable is consistent with the current assignment, then we pass [6.1]

[6.1.1] Give that value to the variable and add it to the assignment.

[6.1.2] Check if the domain set of surrounding variables is decreasing or not.

[6.1.3] If inferences does not meet any variable whose domain set is empty, then go to [6.1.3.1]

[6.1.3.1] Add inferences to the assignment if it has a domain set with only 1 value

[6.1.3.2] Go back to [5], take another variable and continue assigning a value to it.

[6.1.3.3] If result is different from failure then return result

[6.2] If in [6.a] inferences encounters failure state, then we remove {var = value} and inferences from assignment and pass [7].

[7] return false and go to another branch to execute.

10.10.1. Comment on Backtracking:

• Select next variable and value.

+ Select the variable according to the minimum remaining-values heuristic : select the variable with the smallest domain.

+ Select variables according to Degree heuristic: select the variable with the largest number of degrees (degree: the number of constraints that the variables are related to).

+ Choose value according to Least-constraning-value heuristic: choose the variable that leaves us with the most choices for other variables.

• Inference.

+ Forward checking: check the domain of its neighbors.

+ Maintaining arc consistency : check the domain of all variables.

• Backtracking strategies.

+ Backtrack by Backjumping: create a conflict-set set, containing assignments that are likely to conflict with the current variable you are considering.

+ When it encounters a failure, it immediately jumps to the conflict-set set.

10.11. Local search for CSP

10.11.1. Local search and Min-conflicts

• In many CSPs, local search algorithms prove to be effective. They employ a complete-state formulation, where each variable is given a value in the starting state and its value is changed one variable at a time during the search.

• For many CSPs, Min-conflicts is surprisingly successful. Surprisingly, for the n-queens problem, the run time of min-conflicts is essentially independent of problem size if you ignore the initial placement of queens. Even the million-queens puzzle is resolved in an average of 50 steps (after the initial assignment).

• A set of variables, their respective domains, and restrictions on their combined values. A complete assignment to every variable will be a node in the search space. Change the variable's value to breach the constraint quantity.

Input: + A set of variables.

+ A function such that domain (X) is the domain of variable X.

+ Set of constraint to be satisfied the constraints.

Output: Complete assignment that satisfies the constraints.

• For instance: Solve 8-puzzles by using Min-conflicts algorithm.

```
function MIN-CONFLICTS(csp, max_steps) returns a solution or failure
  inputs: csp, a constraint satisfaction problem
          max_steps, the number of steps allowed before giving up

  current ← an initial complete assignment for csp
  for i = 1 to max_steps do
    if current is a solution for csp then return current
    var ← a randomly chosen conflicted variable from csp.VARIABLES
    value ← the value v for var that minimizes CONFLICTS(var, v, current, csp)
    set var = value in current
  return failure
```

Figure 6.8 The MIN-CONFLICTS algorithm for solving CSPs by local search. The initial state may be chosen randomly or by a greedy assignment process that chooses a minimal-conflict value for each variable in turn. The CONFLICTS function counts the number of constraints violated by a particular value, given the rest of the current assignment.

Pseudo code Min-Conflicts algorithm

• Algorithm description:

[1] Input is a CSP consisting of components (X, D, C) and maxsteps and returns a solution (if all variables have been assigned values) or failure.

[2] Input take CSP (a constraint satisfaction problem) and max_step (the number of step before giving up)

[2.1] Initialize the current (an initial complete of steps allowed before giving up).

[3] Traverse a loop running from 1 to max_steps.

> [3.1] Check the *current* is a solution for the csp problem | if true return *current*.
>
> [3.2] Assign the var value to a randomly chosen conflicted variable from csp.Variables.
>
> [3.3] Choose a value v for the variable just initialized above [3.2] such that the minimum number of constraints possible through the function CONFLICTS(var, v, current, csp).
>
> [3.4] set the value of the variable var = value in the *current* sets and keep repeating until the value i = max_steps.

[4] Return failure.

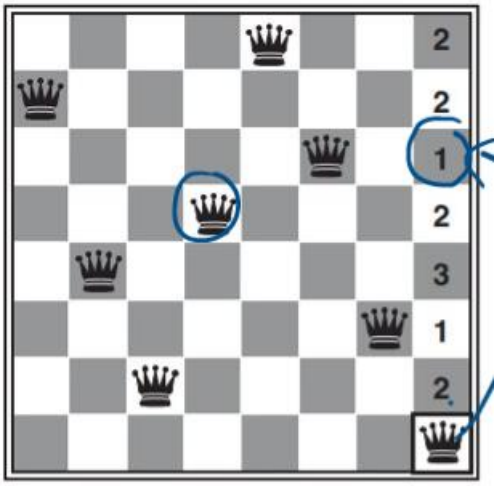
10.11.2. Demo (8-queens)

This is a demo for the problem of 8 queens using min-conflicts algorithm to solve it.

At each stage, a queen is chosen for reassignment in its column. The number of conflicts is show in each square. The algorithm moves the queen to the min-conflict square, breaking ties randomly.

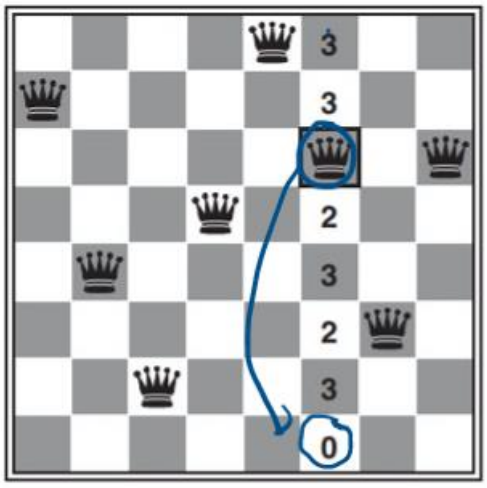
First loop:

We can see that on the first queen, the position of the cells it can move is assigned a number, this number represents the number of constraints if it chooses to stay in that position, so in the algorithm In this math we need to find the positions with the least constraints, so we choose the circled position below



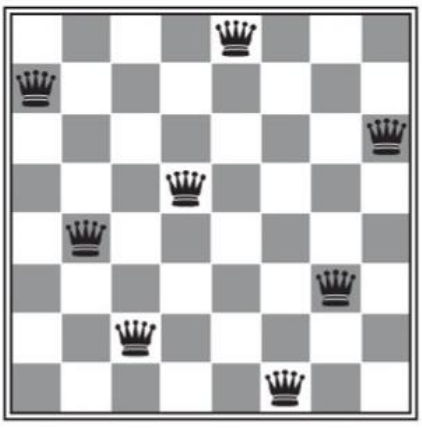
Third loop:

Similar to the first loop above, in this round we see that there is a position where the value is 0 (optimal) because there are no constraints so we put the 3nd queen there.



Solution:

And after running all 8 loops, we have the solution for this 8 queen problem as follows.



10.11.3 Comment on the Min-Conflicts algorithm

• Min-conflicts is surprising effective for many CSPs.

• It solves even the million-queens problem in.

• It has been use to schedule observations for the Humbble Space Telescope, reducing the time taken to schedule a week of observations from.

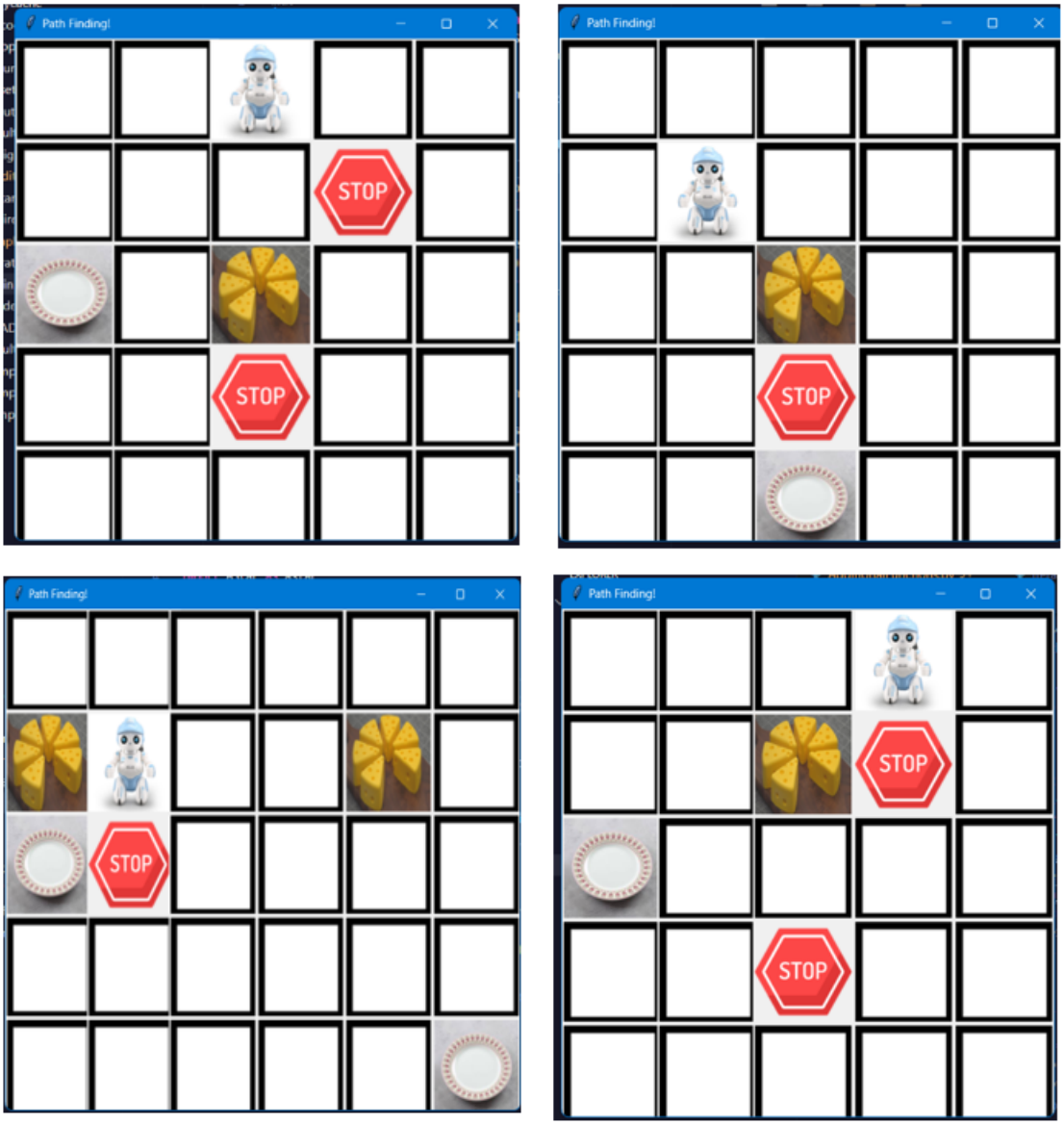
• This algorithm is not completed. (Due to the max_steps).

VI. APPLICATION.

1. Introduction to our project.

In this group project, we have implemented IDS and A* algorithms to find the optimal path. In this program, the robot will push a piece of butter onto its plate and the robot will do this optimally without further movement.

About the interface of the program, we use 6 maps equivalent to 6 levels for the robot to perform: (some example maps).



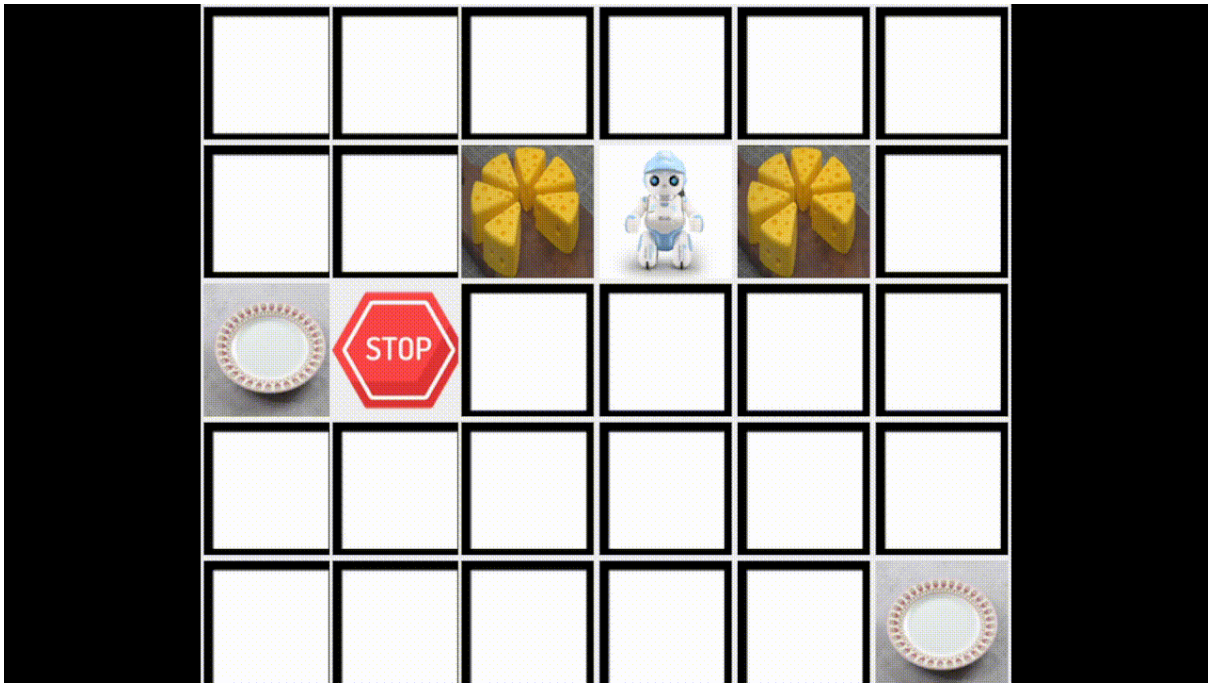
First, we will choose the level for the robot to push the butter on the plate:

```
Which level do you want to use?
1) level 1
2) level 2
3) level 3
4) level 4
5) level 5
6) level 6
```

After that, we will choose the algorithm for the robot to push the butter on the plate:

```
Which Algorithm do you want to use?
1) IDS
2) A*
```

Finally, the robot to push the butter on the plate:



Video demo

The result will include: number of nodes created, number of nodes expanded, duration, path cost, depth of target, path of the algorithm:

```
number of created nodes are: 16995
number of expanded nodes are: 5550
duration is :  8.316758155822754
path costs :  18
depth of goal :  16
path is:  ['l', 'l', 'u', 'l', 'd', 'r', 'r', 'r', 'u', 'r', 'd', 'd', 'd', 'l', 'd', 'r']
************************************************************
```

2. Source Code

2.1. Iterative-deepening search

Firstly, we press key 1 to choose the IDS algorithm for running, then start_time will be set at the current time and finish_time will be set at the time after the IDS algorithm is completed.

```
if __name__ == "__main__":
    file_name = 'test2.txt'
    algorithm = input("Which Algorithm do you want to use?\n1) IDS\n2) BBFS\n3) A*\n")
    if algorithm == "1" or algorithm == "IDS":
        start_time = time.time()
        has_result, path, goal_depth, nodes_info = ids.start_ids_algorithm(file_name, 15)
        finish_time = time.time()
        duration = (finish_time - start_time)
        print("number of created nodes are:", nodes_info[0])
        print("number of expanded nodes are:", nodes_info[1])
        print("duration is : ", duration)

        if not has_result:
            print('there is no answer in this environment!')
        else:
            movement_list = funcs.find_movement_list(path)
            print("path costs : ", path[-1].cost_g)
            print("depth of goal : ", goal_depth)
            print('path is: ', movement_list)
            funcs.print_path(path)
            g = gui.GraphicalInterface(path)
            g.Visualize()
            funcs.write_to_file("IDS", file_name, movement_list, duration)
```

We have the function start_ids_algorithm with parameters that are maps from file_name and max_depth. This function is used to initialize environment and robot_coordinates from function read_file with parameter is map that we choose from the begin.

```
def start_ids_algorithm(test_case_file, max_depth):
    environment = read_file(test_case_file)[1]
    robot_coordinates = read_file(test_case_file)[4]
    all_goal_environment = generate_all_goal_environment(test_case_file)[0]
    root_node = Node(environment, robot_coordinates, 0, "", "", 1, 0)

    result_of_IDS, received_final_state, nodes_info = ids(root_node, all_goal_environment, max_depth)

    if result_of_IDS:
        path = find_path_with_final_node(received_final_state)
        return result_of_IDS, path, received_final_state.depth, nodes_info
    else:
        return result_of_IDS, received_final_state, max_depth, nodes_info
```

2.1.1. Read_file function

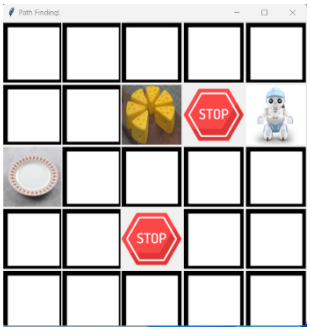
```python
# read file and return information about environment
def read_file(file_name):
    path = "C:\\Users\\HP\\Desktop\\Searching-Algorithms-IDS-BBFS-AStar-main\\input\\" + file_name
    with open(path, 'r') as file:
        size_rows_cols = file.readline()
        number_of_rows = int(size_rows_cols.split("\t")[0])
        number_of_cols = int(size_rows_cols.split("\t")[1])
        number_of_butter = 0

        matrix_with_cost = []
        matrix_cost = []
        matrix_without_cost = []
        robot_coordinates = {"x": 0, "y": 0}
        for i in range(number_of_rows):
            row = file.readline().replace("\t", " ").replace("\n", "").split(" ")

            row_matrix_with_cost, row_matrix_cost, row_matrix_without_cost = [], [], []
            for j in range(number_of_cols):
                row_matrix_with_cost.append(row[j])

                if row[j].isnumeric():
                    row_matrix_without_cost.append("")
                    row_matrix_cost.append(int(row[j]))
                else:
                    if row[j] == "x":
                        row_matrix_without_cost.append(row[j])
                        row_matrix_cost.append(row[j])
                    else:
                        char = row[j][-1]
                        number = row[j][0:-1]
                        row_matrix_without_cost.append(char)
                        row_matrix_cost.append(int(number))
                        if char == "r":
                            robot_coordinates = {"x": i, "y": j}
                        if char == "b":
                            number_of_butter += 1

            matrix_with_cost.append(row_matrix_with_cost)
            matrix_without_cost.append(row_matrix_without_cost)
            matrix_cost.append(row_matrix_cost)

    return matrix_with_cost, matrix_without_cost, matrix_cost, number_of_butter, robot_coordinates
```

In the read_file function, we access to map in the input folder through the path, then read file and return information about the environment such as matrix_with_cost, matrix_without_cost, matrix_cost,_ number_of_butter, and robot_coordinates.

For example:

I choose test2 for the map then:

```
5    5
1    1    1    1    1
1    1    1b   x    1r
2p   1    1    1    1
2    2    x    1    1
1    2    2    2    1
```



Matrix_with_cost will be:

```
'1' '1' '1'  '1' '1'
'1' '1' '1b' 'x' '1r'
'2p' '1' '1' '1' '1'
'2'  '2' 'x' '1' '1'
'1'  '2' '2' '2' '1'
```

Matrix_without_cost will be:

```
''  '' ''  ''  ''
''  '' 'b' 'x' 'r'
'p' '' ''  ''  ''
''  '' 'x' ''  ''
''  '' ''  ''  ''
```

Matrix_cost will be:

```
1  1  1  1  1
1  1  1 'x' 1
2  1  1  1  1
2  2 'x' 1  1
1  2  2  2  1
```

With char p,b,x,r is respectively plates, butter, barrier, and robot so number of butters is 1 and robot_coordinates that is the initial position of robot, is {1,4}.

2.1.2. Generate_all_goal_environment function

```
def generate_all_goal_environment(file_name):
    environment_with_cost, environment_without_cost, environment_cost, number_of_butters, robot_coordinates = read_file(
        file_name)

    start_robot_x_coordinates, start_robot_y_coordinates = robot_coordinates['x'], robot_coordinates['y']
    all_plates_coordinates = find_plates_coordinates(file_name)

    all_butters_coordinates = find_butters_coordinates(file_name)

    for plate_coordinates in all_plates_coordinates:
        plate_x_coordinate, plate_y_coordinate = plate_coordinates['x'], plate_coordinates['y']
        environment_without_cost[plate_x_coordinate][plate_y_coordinate] = "bp"

    for butter_coordinates in all_butters_coordinates:
        butter_x_coordinate, butter_y_coordinate = butter_coordinates['x'], butter_coordinates['y']
        environment_without_cost[butter_x_coordinate][butter_y_coordinate] = ''

    all_final_robot_coordinates = []
    for plate_coordinates in all_plates_coordinates:
        plate_x_coordinate, plate_y_coordinate = plate_coordinates['x'], plate_coordinates['y']
        all_permitted_movements = get_all_permitted_movements(environment_without_cost, plate_coordinates)
        for movement in all_permitted_movements:
            final_robot_coordinates = dsum(plate_coordinates, movement_to_coordinate[movement])

    environment_without_cost[start_robot_x_coordinates][start_robot_y_coordinates] = ""
    all_goal_environment = []
    all_goal_robot_coordinates = []
    for final_robot_coordinates in all_final_robot_coordinates:
        goal_environment = copy.deepcopy(environment_without_cost)
        final_robot_x_coordinates, final_robot_y_coordinates = final_robot_coordinates['x'], final_robot_coordinates[
            'y']
        goal_environment[final_robot_x_coordinates][final_robot_y_coordinates] = 'r'
        all_goal_environment.append(goal_environment)
        all_goal_robot_coordinates.append(final_robot_coordinates)

    return all_goal_environment, all_goal_robot_coordinates
```

This function is used to find all_goal_environment which is all states environment after doing all possible actions from the start_robot_coordinates and all_goal_robot_coordinates is all states of robot_coordinates after executing all possible actions such as down, right, up, or left.

We can find all possible actions by get_all_permitted_movements functions.

2.1.3. Get_all_permitted_movements function

```
# get all permitted movements in each state
def get_all_permitted_movements(environment, robot_coordinates):
    all_movements = {'u', 'r', 'd', 'l'}
    all_permitted_movements = []
    for movement in all_movements:
        if check_next_move(environment, movement, robot_coordinates):
            all_permitted_movements.append(movement)

    return all_permitted_movements
```

In this function, we will have a check_next_move function to check if that move is permitted.

```python
def check_next_move(environment, next_move, robot_coordinates):
    is_first_step_available, robot_next_coordinates = move_forward(environment, next_move, robot_coordinates)
    if not is_first_step_available:
        return False

    else:
        # robot is not allowed to go to the cells with x or bp in it
        if 'bp' in environment[robot_next_coordinates['x']][robot_next_coordinates['y']] or 'x' in \
                environment[robot_next_coordinates['x']][robot_next_coordinates['y']]:
            return False

        if 'b' in environment[robot_next_coordinates['x']][robot_next_coordinates['y']]:
            # robot can't push two cells both with butter or after butter cell, cell is x
            is_two_step_available, butter_next_coordinates = move_forward(environment, next_move,
                                                                          robot_next_coordinates)
            if not is_two_step_available:
                return False

            else:
                if 'b' in environment[butter_next_coordinates['x']][butter_next_coordinates['y']]:
                    return False
                if 'x' in environment[butter_next_coordinates['x']][butter_next_coordinates['y']]:
                    return False
                else:
                    return True

        else:
            return True
```

2.1.4. Check_next_move function

It takes the environment, our next move, and robot coordinates as input and checks whether the robot can make this move or not.

In Generate_all_goal_environment function, we also have a find_plates_coordinates and find_butters_coordinates function which are used to find all plates and butters in our environment.

```python
def find_plates_coordinates(file_name):
    environment_with_cost, environment_without_cost, environment_cost, number_of_butters, robot_coordinates = read_file(
        file_name)
    num_rows, num_cols = len(environment_without_cost), len(environment_without_cost[0])

    plates_coordinates = []
    for i in range(num_rows):
        for j in range(num_cols):
            if environment_without_cost[i][j] == 'p':
                plates_coordinates.append({"x": i, "y": j})

    return plates_coordinates
```

```python
def find_butters_coordinates(file_name):
    environment_with_cost, environment_without_cost, environment_cost, number_of_butters, robot_coordinates = read_file(
        file_name)
    num_rows, num_cols = len(environment_without_cost), len(environment_without_cost[0])

    butters_coordinates = []
    for i in range(num_rows):
        for j in range(num_cols):
            if environment_without_cost[i][j] == 'b':
                butters_coordinates.append({"x": i, "y": j})

    return butters_coordinates
```

2.1.5. IDS function

```
# It uses recursive dls()
def ids(first_node, all_goal_environment, max_depth):
    nodes_info = [0, 0]  # nodes_info[0] is num_of_generates , nodes_info[1] is num_of_expanded
    for depth in range(max_depth):
        goal_node = dls(first_node, all_goal_environment, depth, nodes_info)

        if goal_node:
            return True, goal_node, nodes_info
    return False, "", nodes_info
```

It uses recursive depth limit search (DLS) to check if that problem has a solution or not. If it is true, it will return true to confirm that this problem can be solved and return goal_node and node_info which is the number of created nodes and expanded nodes.

```
# depth limited search
def dls(start_node, all_goal_environment, max_depth, nodes_info):
    nodes_info[1] += 1  # num_of_expand
    if start_node.environment in all_goal_environment:
        return start_node
    if max_depth <= 0:
        return False

    children = create_child(start_node)
    nodes_info[0] += len(children)  # num_of_generate

    for child in children:
        goal_node = dls(child, all_goal_environment, max_depth - 1, nodes_info)
        if goal_node:
            return goal_node
    return False
```

2.1.6. Create child function

```
def create_child(node):
    children_arr = []
    curr_environment, curr_robot_coordinates, curr_depth = node.environment, node.robot_coordinates, node.depth
    all_permitted_movements = get_all_permitted_movements(curr_environment, curr_robot_coordinates)
    for movement in all_permitted_movements:
        new_environment, new_robot_coordinates = update_environment(curr_environment, curr_robot_coordinates, movement)
        child_node = Node(new_environment, new_robot_coordinates, curr_depth + 1, movement, node, node.cost_g + 1, 0)
        children_arr.insert(0, child_node)

    return children_arr
```

This function creates child nodes from a node base on all possible moves which that node can do.

2.1.7. Update environment function

```
# get environment and movement then return new environment and new robot coordinates
def update_environment(environment, current_robot_coordinates, movement):
    new_robot_coordinates = dsum(current_robot_coordinates, movement_to_coordinate[movement])
    curr_robot_x_coordinate, curr_robot_y_coordinate = current_robot_coordinates['x'], current_robot_coordinates['y']
    new_robot_x_coordinate, new_robot_y_coordinate = new_robot_coordinates['x'], new_robot_coordinates['y']

    new_environment = copy.deepcopy(environment)

    # next robot coordinates have butter
    if new_environment[new_robot_x_coordinate][new_robot_y_coordinate] == 'b':
        new_butter_coordinates = dsum(new_robot_coordinates, movement_to_coordinate[movement])
        new_butter_x_coordinate, new_butter_y_coordinate = new_butter_coordinates['x'], new_butter_coordinates['y']

        # next butter coordinates have plate
        if new_environment[new_butter_x_coordinate][new_butter_y_coordinate] == 'p':
            new_environment[new_butter_x_coordinate][new_butter_y_coordinate] = 'bp'
        else:
            new_environment[new_butter_x_coordinate][new_butter_y_coordinate] = 'b'

        new_environment[curr_robot_x_coordinate][curr_robot_y_coordinate] = ''
        new_environment[new_robot_x_coordinate][new_robot_y_coordinate] = 'r'

    # next robot coordinates have plate
        new_environment[curr_robot_x_coordinate][curr_robot_y_coordinate] = ''
        new_environment[new_robot_x_coordinate][new_robot_y_coordinate] = 'rp'

    elif new_environment[new_robot_x_coordinate][new_robot_y_coordinate] == '' and \
            new_environment[curr_robot_x_coordinate][curr_robot_y_coordinate] == 'rp':
        new_environment[curr_robot_x_coordinate][curr_robot_y_coordinate] = 'p'
        new_environment[new_robot_x_coordinate][new_robot_y_coordinate] = 'r'

    else:
        new_environment[curr_robot_x_coordinate][curr_robot_y_coordinate] = ''
        new_environment[new_robot_x_coordinate][new_robot_y_coordinate] = 'r'

    return new_environment, new_robot_coordinates
```

It will get environment and movement then return new environment and new robot coordinates

For instance, if next robot coordinates have butter then a new butter coordinates will be found by dsum function with the movement on new robot coordinate or if next butter coordinates have plate then new butter coordinate will be replace by “bp”.

2.1.8. Find_path_with_final_node function

```
def find_path_with_final_node(node):
    path_by_nodes = []
    pre_node = node

    while pre_node != "":
        path_by_nodes.append(pre_node)
        pre_node = pre_node.parent

    path_by_nodes.reverse()
    return path_by_nodes
```

It will get final node(goal state) and return path from start node to goal node

2.2. Apply A* Algorithm in the Project

2.2.1. Premise functions

There are some library we use in the A* Algorithm include from Node and AdditionalFunction which contains all the essential functions for A* Algorithm

```
from AdditionalFunctions import *
from Node import *
import time
import copy
```

a) AdditionalFunction

****read_file:*** Function for input the map from the file path :

```
# read file and return information about environment
def read_file(file_name):
    path = r"C:\Users\Admin\Desktop\STUDIES\Semester_III_HK1\Artificial Intelligence(AI)\Final_Project\Lựm mạng\New folder (2)\Searching-Algorithms-IDS-BBFS-AStar-main\input
    with open(path, 'r') as file:
        size_rows_cols = file.readline()
        number_of_rows = int(size_rows_cols.split("\t")[0])
        number_of_cols = int(size_rows_cols.split("\t")[1])
        number_of_butter = 0

        matrix_with_cost = []
        matrix_cost = []
        matrix_without_cost = []
        robot_coordinates = {"x": 0, "y": 0}
        for i in range(number_of_rows):
            row = file.readline().replace("\t", " ").replace("\n", "").split(" ")

            row_matrix_with_cost, row_matrix_cost, row_matrix_without_cost = [], [], []
            for j in range(number_of_cols):
                row_matrix_with_cost.append(row[j])

                if row[j].isnumeric():
                    row_matrix_without_cost.append("")
                    row_matrix_cost.append(int(row[j]))
                else:
                    if row[j] == "x":
                        row_matrix_without_cost.append(row[j])
                        row_matrix_cost.append(row[j])
                    else:
                        char = row[j][-1]
                        number = row[j][0:-1]
                        row_matrix_without_cost.append(char)
                        row_matrix_cost.append(int(number))
                        if char == "r":
                            robot_coordinates = {"x": i, "y": j}
                        if char == "b":
                            number_of_butter += 1

            matrix_with_cost.append(row_matrix_with_cost)
            matrix_without_cost.append(row_matrix_without_cost)
            matrix_cost.append(row_matrix_cost)

    return matrix_with_cost, matrix_without_cost, matrix_cost, number_of_butter, robot_coordinates
```

➔The return includes:

+ *matrix_with_cost:* the entire matrix from the file reads into without filtering any characters.

+ *matrix_without_cost:* but the whole position but leave out the number.

+ *matrix_cost:* only number.

+ *number_of_butter*: The amount of butter in the matrix.

+ *robot_coordinates*: Current position of the robot.

For example:

I choose test2 for the map then:

```
5     5
1     1     1     1     1
1     1     1b    x     1r
2p    1     1     1     1
2     2     x     1     1
1     2     2     2     1
```

Matrix_with_cost will be:

```
'1' '1' '1'  '1' '1'
'1' '1' '1b' 'x' '1r'
'2p' '1' '1' '1' '1'
'2'  '2' 'x' '1' '1'
'1'  '2' '2' '2' '1'
```

Matrix_without_cost will be:

```
''  '' ''  ''  ''
''  '' 'b' 'x' 'r'
'p' '' ''  ''  ''
''  '' 'x' ''  ''
''  '' ''  ''  ''
```

Matrix_cost will be:

```
1  1  1  1  1
1  1  1 'x' 1
2  1  1  1  1
2  2 'x' 1  1
1  2  2  2  1
```

With char p,b,x,r is respectively plates, butter, barrier, and robot so number of butters is 1 and robot_coordinates that is the initial position of robot, is {1,4}.

check_next_move : Check function to see if the node point passes or not

```python
# it takes the enviroment, our next move and robot coordinates as input and checks whether the robot can make this
# move or not
def check_next_move(environment, next_move, robot_coordinates):
    is_first_step_available, robot_next_coordinates = move_forward(environment, next_move, robot_coordinates)
    if not is_first_step_available:
        return False

    else:
        # robot is not allowed to go to the cells with x or bp in it
        if 'bp' in environment[robot_next_coordinates['x']][robot_next_coordinates['y']] or 'x' in \
                environment[robot_next_coordinates['x']][robot_next_coordinates['y']]:
            return False

        if 'b' in environment[robot_next_coordinates['x']][robot_next_coordinates['y']]:
            # robot can't push two cells both with butter or after butter cell, cell is x
            is_two_step_available, butter_next_coordinates = move_forward(environment, next_move,
                                                                          robot_next_coordinates)
            if not is_two_step_available:
                return False

            else:
                if 'b' in environment[butter_next_coordinates['x']][butter_next_coordinates['y']]:
                    return False
                if 'x' in environment[butter_next_coordinates['x']][butter_next_coordinates['y']]:
                    return False
                else:
                    return True

        else:
            return True
```

```python
def find_path_with_final_node(node):
    path_by_nodes = []
    pre_node = node

    while pre_node != "":
        path_by_nodes.append(pre_node)
        pre_node = pre_node.parent

    path_by_nodes.reverse()
    return path_by_nodes
```

Find_path_with_final_node: It will get final node(goal state) and return path from start node to goal node

get_all_permitted_movements : Returns the movements that can go (are the movements where k entangles or x)

```python
# get all permitted movements in each state
def get_all_permitted_movements(environment, robot_coordinates): #Trả về những movements có thể đi được ( là những movement mà k vướng butter hay là x )
    all_movements = {'u', 'r', 'd', 'l'}
    all_permitted_movements = []
    for movement in all_movements:
        if check_next_move(environment, movement, robot_coordinates):
            all_permitted_movements.append(movement)

    return all_permitted_movements
```

move_forward : Function to make a move to a new node and return the new location of the robot

```python
def move_forward(environment, next_move, robot_coordinates): #Di chuyển
    num_rows, num_cols = len(environment), len(environment[0])
    new_robot_coordinates = dsum(robot_coordinates, movement_to_coordinate[next_move])

    # robot is not allowed to go outside the environment
    if 0 <= new_robot_coordinates['x'] < num_rows and 0 <= new_robot_coordinates['y'] < num_cols:
        return True, new_robot_coordinates
    else:
        return False, new_robot_coordinates
```

**** find_plates_coordinates*** : Find the location of plates in the matrix

```python
def find_plates_coordinates(file_name):
    environment_with_cost, environment_without_cost, environment_cost, number_of_butters, robot_coordinates = read_file(
        file_name)
    num_rows, num_cols = len(environment_without_cost), len(environment_without_cost[0])

    plates_coordinates = []
    for i in range(num_rows):
        for j in range(num_cols):
            if environment_without_cost[i][j] == 'p':
                plates_coordinates.append({"x": i, "y": j})

    return plates_coordinates
```

**** find_butters_coordinates*** : Find the location of the butters in the matrix

```python
def find_butters_coordinates(file_name):
    environment_with_cost, environment_without_cost, environment_cost, number_of_butters, robot_coordinates = read_file(
        file_name)
    num_rows, num_cols = len(environment_without_cost), len(environment_without_cost[0])

    butters_coordinates = []
    for i in range(num_rows):
        for j in range(num_cols):
            if environment_without_cost[i][j] == 'b':
                butters_coordinates.append({"x": i, "y": j})

    return butters_coordinates
```

**** generate_all_goal_environment :*** This function is used to find all_goal_environment which is all states environment after doing all possible actions from the start_robot_coordinates and all_goal_robot_coordinates is all states of robot_coordinates after executing all possible actions such as down, right, up, or left.

We can find all possible actions by get_all_permitted_movements functions*

```python
def generate_all_goal_environment(file_name):
    environment_with_cost, environment_without_cost, environment_cost, number_of_butters, robot_coordinates = read_file(
        file_name)

    start_robot_x_coordinates, start_robot_y_coordinates = robot_coordinates['x'], robot_coordinates['y']
    all_plates_coordinates = find_plates_coordinates(file_name) # tìm vị trí của toàn bộ đĩa trong tập

    all_butters_coordinates = find_butters_coordinates(file_name) # tìm vị trí của toàn bộ bơ trong tập

    for plate_coordinates in all_plates_coordinates:
        plate_x_coordinate, plate_y_coordinate = plate_coordinates['x'], plate_coordinates['y']
        environment_without_cost[plate_x_coordinate][plate_y_coordinate] = "bp"

    for butter_coordinates in all_butters_coordinates:
        butter_x_coordinate, butter_y_coordinate = butter_coordinates['x'], butter_coordinates['y']
        environment_without_cost[butter_x_coordinate][butter_y_coordinate] = ''

    all_final_robot_coordinates = []
    for plate_coordinates in all_plates_coordinates:
        plate_x_coordinate, plate_y_coordinate = plate_coordinates['x'], plate_coordinates['y']
        all_permitted_movements = get_all_permitted_movements(environment_without_cost, plate_coordinates)
        for movement in all_permitted_movements:
            final_robot_coordinates = dsum(plate_coordinates, movement_to_coordinate[movement])
            all_final_robot_coordinates.append(final_robot_coordinates) # lưu lại đường đi cuối cùng

    environment_without_cost[start_robot_x_coordinates][start_robot_y_coordinates] = ""
    all_goal_environment = []
    all_goal_robot_coordinates = []
    for final_robot_coordinates in all_final_robot_coordinates:
        goal_environment = copy.deepcopy(environment_without_cost)
        final_robot_x_coordinates, final_robot_y_coordinates = final_robot_coordinates['x'], final_robot_coordinates[
            'y']
        goal_environment[final_robot_x_coordinates][final_robot_y_coordinates] = 'r'
        all_goal_environment.append(goal_environment)
        all_goal_robot_coordinates.append(final_robot_coordinates)

    return all_goal_environment, all_goal_robot_coordinates
```

update_environment : Update the map after each robot move, butter

```python
# get environment and movement then return new environment and new robot coordinates
def update_environment(environment, current_robot_coordinates, movement):
    new_robot_coordinates = dsum(current_robot_coordinates, movement_to_coordinate[movement])
    curr_robot_x_coordinate, curr_robot_y_coordinate = current_robot_coordinates['x'], current_robot_coordinates['y']
    new_robot_x_coordinate, new_robot_y_coordinate = new_robot_coordinates['x'], new_robot_coordinates['y']

    new_environment = copy.deepcopy(environment)

    # next robot coordinates have butter
    if new_environment[new_robot_x_coordinate][new_robot_y_coordinate] == 'b':
        new_butter_coordinates = dsum(new_robot_coordinates, movement_to_coordinate[movement])
        new_butter_x_coordinate, new_butter_y_coordinate = new_butter_coordinates['x'], new_butter_coordinates['y']

        # next butter coordinates have plate
        if new_environment[new_butter_x_coordinate][new_butter_y_coordinate] == 'p':
            new_environment[new_butter_x_coordinate][new_butter_y_coordinate] = 'bp'
        else:
            new_environment[new_butter_x_coordinate][new_butter_y_coordinate] = 'b'

        new_environment[curr_robot_x_coordinate][curr_robot_y_coordinate] = ''
        new_environment[new_robot_x_coordinate][new_robot_y_coordinate] = 'r'

    # next robot coordinates have plate
    elif new_environment[new_robot_x_coordinate][new_robot_y_coordinate] == 'p':
        new_environment[curr_robot_x_coordinate][curr_robot_y_coordinate] = ''
        new_environment[new_robot_x_coordinate][new_robot_y_coordinate] = 'rp'

    elif new_environment[new_robot_x_coordinate][new_robot_y_coordinate] == '' and \
            new_environment[curr_robot_x_coordinate][curr_robot_y_coordinate] == 'rp':
        new_environment[curr_robot_x_coordinate][curr_robot_y_coordinate] = 'p'
        new_environment[new_robot_x_coordinate][new_robot_y_coordinate] = 'r'

    else:
        new_environment[curr_robot_x_coordinate][curr_robot_y_coordinate] = ''
        new_environment[new_robot_x_coordinate][new_robot_y_coordinate] = 'r'

    return new_environment, new_robot_coordinates
```

b)Node:

In this Node program, contains the Node class and the new node initialization function :

```python
class Node:
    def __init__(self, environment, robot_coordinates, depth, movement, parent, cost_g, cost_f):
        self.environment = environment
        self.robot_coordinates = robot_coordinates
        self.depth = depth
        self.movement = movement
        self.parent = parent
        self.cost_g = cost_g
        self.cost_f = cost_f


class Initial_node:
    def __init__(self, environment):
        self.depth = -1
        self.environment = environment
```

In this node class include:

+ *environment* : Contains the current map.

+ *robot_coordinates* : The current position of the robot.

+ *depth:* The current depth of the node in the path.

+ *movement:* Returns the movements that the current node has permission to go.

+ *parent:* The parent node, also known as the previous node of the current node to find the path and calculate the cost path.

+ *cost_g:* Path cost from initial node to current node.

+ *cost_f:* Path cost from initial node to current node + heuristic from current point to goal node.

2.3. Algorithms implementation Program

calculate_manhattan_distance : Calculate the distance from point 1 to point 2 based on the location of 2 points

```
def calculate_manhattan_distance(point1, point2): #tính khoảng cách từ point 1 đến point 2
    x_point1, y_point1 = point1['x'], point1['y']
    x_point2, y_point2 = point2['x'], point2['y']

    return abs(x_point1 - x_point2) + abs(y_point1 - y_point2)
```

calculate_heuristic_environment : function returns a map each element of which is the distances from that point to plates

```
def calculate_heuristic_environment(plates_coordinates_arr, environment_without_cost):
    num_rows, num_cols = len(environment_without_cost), len(environment_without_cost[0])
    heuristic_environment = [[0 for i in range(num_cols)] for j in range(num_rows)]

    for i in range(num_rows):
        for j in range(num_cols):
            heuristic_environment[i][j] = calculate_distance_point_to_all_plates({"x": i, "y": j},
                                                                                 plates_coordinates_arr)

    return heuristic_environment
```

generate_node : create node and input the essential information for this node and add this node to the frontier array and explored array

```
def generate_node(node, envrironment_cost, hueristic_arr, frontier, explored):
    curr_environment, curr_robot_coordinates, curr_depth = node.environment, node.robot_coordinates, node.depth
    curr_cost_g, curr_cost_f = node.cost_g, node.cost_f
    all_permitted_movements = get_all_permitted_movements(curr_environment, curr_robot_coordinates)
    num_of_generate = len(all_permitted_movements)
    for movement in all_permitted_movements:
        new_environment, new_robot_coordinates = update_environment(curr_environment, curr_robot_coordinates, movement)
        new_cost_g = calculate_cost_g_node(envrironment_cost, new_robot_coordinates, curr_cost_g)
        new_cost_f = calculate_cost_f_node(new_cost_g, new_robot_coordinates, hueristic_arr)
        new_node = Node(new_environment, new_robot_coordinates, curr_depth + 1, movement, node, new_cost_g, new_cost_f)
        update_frontier_explored(frontier, explored, new_node)

    return num_of_generate
```

* ***find_closest_plate*** : Find the closest plate from the current location

```
def find_closest_plate(now_coordinates, plates_coordinates): # hàm tìm đường đến chiếc đĩa nhanh nhất
    distance_arr = []
    for plate_coordinates in plates_coordinates:
        distance_arr.append(calculate_manhattan_distance(now_coordinates, plate_coordinates))

    best_plate_index = distance_arr.index(min(distance_arr))
    best_plate = plates_coordinates[best_plate_index]
    return best_plate
```

* ***calculate_distance_point_to_all_plates*** : calculate the distance from a point to the closest plates

```
def calculate_distance_point_to_all_plates(point, plates_coordinates_arr):
    distance = 0
    point_copy = point
    plates_coordinates_arr_copy = copy.deepcopy(plates_coordinates_arr)
    while plates_coordinates_arr_copy:
        best_plate = find_closest_plate(point_copy, plates_coordinates_arr_copy)
        plates_coordinates_arr_copy.remove(best_plate)
        distance += calculate_manhattan_distance(point_copy, best_plate)

        point_copy = best_plate


    return distance
```

sort_frontier_by_cost : Sort the frontiers based on cost_f properties*

```
# sort frontier list by cost nodes
def sort_frontier_by_cost(frontier):
    frontier.sort(key=lambda x: x.cost_f, reverse=False)
```

* ***calculate_cost_f_node*** : Caculate H(n) function from total path cost and huerisitic

```python
# return cost f of point
def calculate_cost_f_node(new_cost_g, new_robot_coordinates, huerisitic_arr): # hàm tính tổng heristic và cost path từ điểm mới
    return int(new_cost_g) + huerisitic_arr[new_robot_coordinates['x']][new_robot_coordinates['y']]
```

calculate_cost_g_node : Caculate Path cost function from initial state to current state

```python
#return cost g of point
def calculate_cost_g_node(environment_cost, new_robot_coordinates, curr_cost_g): # hàm tính tổng heristic và cost path từ điểm hiện tại
    return int(curr_cost_g) + int(environment_cost[new_robot_coordinates['x']][new_robot_coordinates['y']])
```

**** is_new_node_in_frontier*** : This function to Check whether if the new node that are already in the frontier array or not

```python
def is_new_node_in_frontier(frontier, new_node):
    for node in frontier:
        if node.environment == new_node.environment:
            return True, node
    return False, new_node
```

**** is_new_node_in_explored*** : This function to Check whether if the new node that are already in the explored array or not

```python
def is_new_node_in_explored(explored, new_node):
    for node in explored:
        if node.environment == new_node.environment:
            return True
    return False
```

update_frontier_explored : This function to Check whether if the new node that are in the explored array is in the frontier array for avoiding infinity loop and currently sort the frontier for choosing the best node to move to

```python
def update_frontier_explored(frontier, explored, new_node):
    if not is_new_node_in_explored(explored, new_node):
        is_duplicate, duplicate_node = is_new_node_in_frontier(frontier, new_node)
        if is_duplicate:
            if new_node.cost_f < duplicate_node.cost_f:
                frontier.remove(duplicate_node)
                frontier.append(new_node)
        else:
            frontier.append(new_node)

    sort_frontier_by_cost(frontier)
```

**** generate_node*** : This Function is for updating the current node location in the map and creating new nodes from the current node transmitted and add the number of nodes by number of movements, also known as successors from the current position of the robot. Then update the created nodes into the frontier and explored set *

```
def generate_node(node, envrironment_cost, hueristic_arr, frontier, explored):
    curr_environment, curr_robot_coordinates, curr_depth = node.environment, node.robot_coordinates, node.depth
    curr_cost_g, curr_cost_f = node.cost_g, node.cost_f
    all_permitted_movements = get_all_permitted_movements(curr_environment, curr_robot_coordinates)
    num_of_generate = len(all_permitted_movements)
    for movement in all_permitted_movements:
        new_environment, new_robot_coordinates = update_environment(curr_environment, curr_robot_coordinates, movement)
        new_cost_g = calculate_cost_g_node(envrironment_cost, new_robot_coordinates, curr_cost_g)
        new_cost_f = calculate_cost_f_node(new_cost_g, new_robot_coordinates, hueristic_arr)
        new_node = Node(new_environment, new_robot_coordinates, curr_depth + 1, movement, node, new_cost_g, new_cost_f)
        update_frontier_explored(frontier, explored, new_node)

    return num_of_generate
```

**** expanding_node*** : Function to explore node from the node which input into this function

```
def expanding_node(node, environment_cost, hueristic_arr, frontier, explored):
    num_of_generate = generate_node(node, environment_cost, hueristic_arr, frontier, explored)
    explored.append(node)
    return num_of_generate
```

2.3.1 A* algorithm

The main Algorithm function A* applying to the game

Starting a loop run until the solution path is found or coming to max_depth and still not find the solution. In that loop :

+) First, check whether the current map or environment is the goal environment, if so, return true and return goal_state = current_state

+) Otherwise, expand node with frontier[0] (i.e. which frontier has the highest cost_f, expand that frontier) and update the environment after each node expansion until the environment in the current state is equal to goal_environment, return goal_state. If you run along this path without reaching the solution, if there are no more nodes in the frontier set, you have explored all the nodes and returned no solution .

```python
def a_star_algorithm(start_node, max_depth, envrironment_cost, hueristic_arr, all_goal_environment):
    frontier, explored = [start_node], []
    expand_node = start_node
    is_receive, goal_state = False, 0
    num_of_generate = 0
    num_of_expand = 0

    while expand_node.depth <= max_depth:
        if expand_node.environment in all_goal_environment:
            is_receive, goal_state = True, expand_node
            break
        generate = expanding_node(expand_node, envrironment_cost, hueristic_arr, frontier, explored)
        num_of_generate += generate
        num_of_expand += 1
        frontier.remove(expand_node)

        if len(frontier) == 0:
            break
        expand_node = frontier[0]

    if is_receive:
        return True, goal_state, [num_of_generate, num_of_expand]
    else:
        return False, "can't pass the butter", [num_of_generate, num_of_expand]
```

**** start_a_star_algorithm*** : Full function to be able to run – to be called from main function

- This function initializes environments from the input file such as environment_without_cost, environment_cost , heuristic (environment in which each element is the distance from the part that to the finish plate) and goal_environment when completed. They also caculate the root_cost_g and root_cost_f thank to the above environments .Then use that attributes to create root_node for a_star_algorithm function.

Passing the essential parameter to the a_star_algorithm function to run and return :

+) *result_of_a_star_algorithm* : check variable to see if the solution is found or not.

+) *path* : contains the path of the robot that has gone.

+) *received_final_state .depth* : contains the depth of the journey that the robot has traveled.

+) *nodes_info* : Node parameters such as the number of nodes created and expanded.

```python
def start_a_star_algorithm(test_case_file, max_depth):
    environment_without_cost = read_file(test_case_file)[1] # đọc toàn bộ chướng ngại và bơ và plate
    environment_cost = read_file(test_case_file)[2]
    robot_coordinates = read_file(test_case_file)[4]
    all_goal_environment = generate_all_goal_environment(test_case_file)[0]
    arr_plates = find_plates_coordinates(test_case_file) # trả về tập (x,y) của các plate
    heuristic = calculate_heuristic_environment(arr_plates, environment_without_cost) #Trả về mảng chứa các cái cost từ điểm đó đến plates
    root_cost_g = environment_cost[robot_coordinates['x']][robot_coordinates['y']] # vị trí hiện
    root_cost_f = heuristic[robot_coordinates['x']][robot_coordinates['y']] #
    root_node = Node(environment_without_cost, robot_coordinates, 0, "", "", root_cost_g, root_cost_f)

    result_of_a_star_algorithm, received_final_state, nodes_info = a_star_algorithm(root_node, max_depth, environment_cost,
                                                                                    heuristic, all_goal_environment)
    if result_of_a_star_algorithm:
        path = find_path_with_final_node(received_final_state)
        return result_of_a_star_algorithm, path, received_final_state.depth, nodes_info
    else:
        return result_of_a_star_algorithm, received_final_state, max_depth, nodes_info
```

***Calling algorithm A* in the main function**

```
elif algorithm == "3" or algorithm == "A*":
    start_time = time.time()
    has_result, path, goal_depth, nodes_info = astar.start_a_star_algorithm(file_name, 20)
    finish_time = time.time()
    duration = (finish_time - start_time)

    print("number of created nodes are:", nodes_info[0])
    print("number of expanded nodes are:", nodes_info[1])
    print("duration is : ", duration)

    if not has_result:
        print('there is no answer in this environment!')
    else:
        movement_list = funcs.find_movement_list(path)
        print("path costs : ", path[-1].cost_g)
        print("depth of goal : ", goal_depth)
        print('path is: ', movement_list)
        funcs.print_path(path)
        g = gui.GraphicalInterface(path)
        g.Visualize()
        funcs.write_to_file("ASTAR", file_name, movement_list, duration)
```

REFERENCES

[1] Stuart J. Russell, P. N. (2016). *Artificial Intelligence A Modern Approach*. EdinburghGate Harlow Essex England: Pearson Education © 2010.

[2] Lasse Schultebraucks. (n.d.). *A Short History of Artificial Intelligence* (Dec 5, 2018). Retrieved 04/12/2022 from https://dev.to/lschultebraucks/a-short-history-of-artificial-intelligence-7hm

[3] *Artificial Intelligence* (n.d.). Wikipedia, the free encyclopedia. Retrieve 04/12/2022 from https://wikipedia.org/wiki/Artificial_intelligence

[4] PhD.Tran Nhat Quang (2022). *Lectures on Artificial Intelligence*

[5] *Hill climbing algorithm in AI -Javatpoint.*(n.d.).www.javatpoint.com Retrieve 05/12/2022 from https://www.javatpoint.com/hill-climbing-algorithm-in-ai

[6] AryanRaja (n.d.). *Iterative deepening Search* (July 26, 2022). Retrieve 06/12/2022 from https://www.geeksforgeeks.org/iterative-deepening-searchids-iterative-deepening-depth-first-searchiddfs/

[7] *A* search algorithm*. (October 2, 2022). Wikipedia, the free encyclopedia. Retrieved 07/12/2022, from https://en.wikipedia.org/wiki/A*_search_algorithm

[8] Joshua Willman (n.d.). *Create a GUI Using Tkinter and Python* (August 16, 2022). Retrieve 09/12/2022 from https://www.pythonguis.com/tutorials/create-gui-tkinter/