Lecture Note: Introduction to Artificial Intelligence
Bài Giảng: Nhập Môn Trí Tuệ Nhân Tạo

Nguyễn Quản Bá Hồng*

Ngày 10 tháng 6 năm 2025

Tóm tắt nội dung

This text is a part of the series *Some Topics in Advanced STEM & Beyond*:
URL: https://nqbh.github.io/advanced_STEM/.
Latest version:

- *Lecture Note: Introduction to Artificial Intelligence – Bài Giảng: Nhập Môn Trí Tuệ Nhân Tạo.*
  PDF: URL: https://github.com/NQBH/advanced_STEM_beyond/blob/main/AI/lecture/NQBH_introduction_AI_lecture.pdf.
  TEX: URL: https://github.com/NQBH/advanced_STEM_beyond/blob/main/AI/lecture/NQBH_introduction_AI_lecture.tex.
- *Codes*:
  - C++: https://github.com/NQBH/advanced_STEM_beyond/tree/main/AI/C++.
  - Python: https://github.com/NQBH/advanced_STEM_beyond/tree/main/AI/Python.

Mục lục

*A scientist- & creative artist wannabe, a mathematics & computer science lecturer of Department of Artificial Intelligence & Data Science (AIDS), School of Technology (SOT), UMT Trường Đại học Quản lý & Công nghệ TP.HCM, Hồ Chí Minh City, Việt Nam.
E-mail: nguyenquanbahong@gmail.com & hong.nguyenquanba@umt.edu.vn. Website: https://nqbh.github.io/. GitHub: https://github.com/NQBH.

1 Some Basic Concepts – Vài Khái Niệm Cơ Bản

Phần này giới thiệu 1 số bài toán thường gặp Khoa Học Máy Tính nói chung, & trong AI nói riêng, e.g., bài toán tìm kiếm (search problem), bài toán tối ưu (optimization problem), bài toán tìm đường đi ngắn nhất (shortest path problem).

1.1 Search problems – Bài toán tìm kiếm

Resources – Tài nguyên.

1. Wikipedia/search problem.

In computational complexity theory & computability theory, a *search problem* is a computational problem of finding an *admissible* answer for a given input value, provided that such an answer exists. In fact, a search problem is specified by a binary relation R where xRy iff "y is an admissible answer given x". Search problems frequently occur in graph theory & combinatorial optimization, e.g. searching for matchings, optional cliques, & stable sets in a given undirected graph.

– Trong lý thuyết độ phức tạp tính toán và lý thuyết khả năng tính toán, 1 bài toán tìm kiếm là 1 bài toán tính toán tìm 1 câu trả lời có thể chấp nhận được cho 1 giá trị đầu vào nhất định, với điều kiện là câu trả lời như vậy tồn tại. Trên thực tế, 1 bài toán tìm kiếm được chỉ định bởi 1 quan hệ nhị phân R trong đó xRy nếu và chỉ nếu "y là 1 câu trả lời có thể chấp nhận được cho x". Các bài toán tìm kiếm thường xảy ra trong lý thuyết đồ thị và tối ưu hóa tổ hợp, ví dụ như tìm kiếm các phép khớp, các nhóm tùy chọn và các tập ổn định trong 1 đồ thị vô hướng nhất định.

An algorithm is said to solve a search problem if, for every input value x, it returns an admissible answer y for x when such an answer exists; otherwise, it returns any appropriate output, e.g., "not found" for x with no such answer.

Definition 1 (Search problem). *If R is a binary relation s.t. $\text{field}(R) \subseteq \Gamma^+$ & T is a Turing machine, then T calculates f if:*

- *if x is s.t. there is some y s.t. $R(x, y)$ then T accepts x with output z s.t. $R(x, z)$ (there may be multiple y, & T need only find 1 of them).*
- *If x is s.t. there is no y s.t. $R(x, y)$ then T rejects x.*

The graph of a partial function is a binary relation, & if T calculates a partial function then there is at most 1 possible output.

A R can be viewed as a *search problem*, & a Turing machine which calculates R is also said to solve it. Every search problem has a corresponding decision problem, namely $L(R) = \{x; \exists y,\ R(x, y)\}$. This definition can be generalized to n-ary relations by any suitable encoding which allows multiple strings to be compressed into 1 string (e.g., by listing them consecutively with a delimiter).

1.2 Optimization problem – Bài toán tối ưu

Resources – Tài nguyên.

1. Wikipedia/optimization problem.

In mathematics, engineering, cS, & economics, an *optimization problem* is the computational problem of finding the *best* solution from all feasible solutions.

– Trong toán học, kỹ thuật, khoa học máy tính và kinh tế, bài toán tối ưu hóa là bài toán tìm giải pháp tốt nhất từ tất cả các giải pháp khả thi.

Optimization problems can be divided into 2 categories, depending on whether the variables are continuous or discrete:

1. An optimization problem with discrete variables is known as a discrete optimization, in which an athematical object e.g. an integer, permutations or graph must be found from a countable set.
2. A problem with continuous variables is known as a continuous optimization, in which an optimal value from a continuous function must be found. They can include constrained problems & multimodal problems.

– Các bài toán tối ưu hóa có thể được chia thành 2 loại, tùy thuộc vào việc các biến là liên tục hay rời rạc:

- 1 bài toán tối ưu hóa với các biến rời rạc được gọi là tối ưu hóa rời rạc, trong đó 1 đối tượng như số nguyên, hoán vị hoặc đồ thị phải được tìm thấy từ 1 tập đếm được.
- 1 bài toán với các biến liên tục được gọi là tối ưu hóa liên tục, trong đó 1 giá trị tối ưu từ 1 hàm liên tục phải được tìm thấy. Chúng có thể bao gồm các bài toán bị ràng buộc và các bài toán đa phương thức.

1.2.1 Search space – Không gian tìm kiếm

In the context of an optimization problem, the *search space* refers to the set of all possible points or solutions that satisfy the problem's constraints, targets, or goals. These points represent the feasible solutions that can be evaluated to find the optimal solution according to the objective function. The search space is often defined by the domain of the function being optimized, encompassing all valid inputs that meet the problem's requirements.

– Trong bối cảnh của 1 bài toán tối ưu hóa, *không gian tìm kiếm* đề cập đến tập hợp tất cả các điểm hoặc giải pháp có thể thỏa mãn các ràng buộc, mục tiêu hoặc mục đích của bài toán. Các điểm này biểu thị các giải pháp khả thi có thể được đánh giá để tìm ra giải pháp tối ưu theo hàm mục tiêu. Không gian tìm kiếm thường được xác định theo miền của hàm đang được tối ưu hóa, bao gồm tất cả các đầu vào hợp lệ đáp ứng các yêu cầu của bài toán.

The search space can vary significantly in size & complexity depending on the problem. E.g., in a continuous optimization problem, the search space might be a multidimensional real-valued domain defined by bounds or constraints. In a discrete optimization problem, e.g. combinatorial optimization, the search space could consist of a finite set of permutations, combinations, or configurations.

– Không gian tìm kiếm có thể thay đổi đáng kể về kích thước & độ phức tạp tùy thuộc vào vấn đề. Ví dụ, trong 1 vấn đề tối ưu hóa liên tục, không gian tìm kiếm có thể là 1 miền giá trị thực đa chiều được xác định bởi các giới hạn hoặc ràng buộc. Trong 1 vấn đề tối ưu hóa rời rạc, ví dụ như tối ưu hóa tổ hợp, không gian tìm kiếm có thể bao gồm 1 tập hợp hữu hạn các hoán vị, tổ hợp hoặc cấu hình.

In some contexts, the term *search space* may also refer to the optimization of the domain itself, e.g. determining the most appropriate set of variables or parameters to define the problem. Understanding & effectively navigating the search space is crucial for designing efficient algorithms, as it directly influences the computational complexity & the likelihood of finding an optimal solution.

– Trong 1 số ngữ cảnh, thuật ngữ *search space* cũng có thể đề cập đến việc tối ưu hóa chính miền đó, ví dụ: xác định tập hợp các biến hoặc tham số phù hợp nhất để xác định vấn đề. Hiểu & điều hướng hiệu quả không gian tìm kiếm là rất quan trọng để thiết kế các thuật toán hiệu quả, vì nó ảnh hưởng trực tiếp đến độ phức tạp tính toán & khả năng tìm ra giải pháp tối ưu.

1.2.2 Continuous optimization problem – Bài toán tối ưu liên tục

The standard/canonical form of a continuous optimization problem is

$$\min_x f(x) \text{ subject to } \begin{cases} g_i(x) \leq 0, \ \forall i \in [m], \\ h_j(x) = 0, \ \forall j \in [p], \end{cases}$$

where

- $f : \mathbb{R}^n \to \mathbb{R}$: objective function to be minimized over the n-variable vector x
- $g_i(x) \leq 0$ are called *inequality constraints*
- $h_j(x) = 0$ are called *equality constraints*
- $m \geq 0, p \geq 0$.

If $m = p = 0$, the problem is an unconstrained optimization problem. By convention, the standard form defines a *minimization problem*. A *maximization problem* can be treated by negating the objective function.

– Nếu $m = p = 0$, bài toán là bài toán tối ưu hóa không bị ràng buộc. Theo quy ước, dạng chuẩn định nghĩa 1 *bài toán tối thiểu hóa*. 1 *bài toán tối đa hóa* có thể được xử lý bằng cách phủ định hàm mục tiêu.

1.2.3 Combinatorial optimization problem – Bài toán tối ưu tổ hợp

Formally, a combinatorial optimization problem A is a quadruple (I, f, m, g), where

- I is a set of instances
- given an instance $x \in I$, $f(x)$ is the set of feasible solutions
- given an instance x & a feasible solution y of x, $m(x, y)$ denotes the measure of y, which is usually a positive real.
- g: goal function, & is either min or max

The goal is then to find for some instance x an *optimal solution*, i.e., a feasible solution y with $m(x, y) = g\{m(x, y') : y' \in f(x)\}$.

For each combinatorial optimization problem, there is a corresponding decision problem that asks whether there is a feasible solution for some particular measure m_0. E.g., if there is a graph G which contains vertices u, v, an optimization problem might be "find a path from u to v that uses the fewest edges". This problem might have an answer of, say, 4. A corresponding decision problem would be "is there a path from u, v that uses 10 or fewer edges?" This problem can be answered with a simple `yes` or `no`.

In the field of approximation algorithms, algorithms are designed to find near-optimal solutions to hard problems. The usual decision version is then an inadequate definition of the problem since it only specifies acceptable solutions. Even though we could introduce suitable decision problems, the problem is more naturally characterized as an optimization problem.

– Trong lĩnh vực thuật toán xấp xỉ, thuật toán được thiết kế để tìm ra các giải pháp gần tối ưu cho các vấn đề khó. Phiên bản quyết định thông thường sau đó là 1 định nghĩa không đầy đủ về vấn đề vì nó chỉ nêu ra các giải pháp chấp nhận được. Mặc dù chúng ta có thể đưa ra các vấn đề quyết định phù hợp, nhưng vấn đề này được đặc trưng tự nhiên hơn là 1 vấn đề tối ưu hóa.

For broader coverage of this topic, see Wikipedia/mathematical optimization, also called *mathematical programming*.

1.3 Search algorithms – Các thuật toán tìm kiếm

Resources – Tài nguyên.

1. Wikipedia/search algorithm.

In CS, a *search algorithm* is an algorithm designed to solve a search problem. Search algorithms work to retrieve information stored within particular data structure, or calculated in the search space/feasible region of a problem domain, with either discrete or continuous values.

– Trong khoa học máy tính, thuật toán tìm kiếm là thuật toán được thiết kế để giải quyết vấn đề tìm kiếm. Thuật toán tìm kiếm hoạt động để truy xuất thông tin được lưu trữ trong cấu trúc dữ liệu cụ thể hoặc được tính toán trong không gian tìm kiếm của miền vấn đề, với các giá trị rời rạc hoặc liên tục.

Although search engines use search algorithms, they belong to the study of information retrieval, not algorithmics.

– Mặc dù công cụ tìm kiếm sử dụng thuật toán tìm kiếm nhưng chúng thuộc về lĩnh vực nghiên cứu về truy xuất thông tin chứ không phải thuật toán.

The appropriate search algorithm to use often depends on the data structure being searched, & may also include prior knowledge about the data. Search algorithms can be made faster or more efficient by specially constructed database structures, e.g. search trees, hash maps, & database indexes.

– Thuật toán tìm kiếm phù hợp để sử dụng thường phụ thuộc vào cấu trúc dữ liệu đang được tìm kiếm và cũng có thể bao gồm kiến thức trước đó về dữ liệu. Thuật toán tìm kiếm có thể được thực hiện nhanh hơn hoặc hiệu quả hơn bằng các cấu trúc cơ sở dữ liệu được xây dựng đặc biệt, chẳng hạn như cây tìm kiếm, bản đồ băm và chỉ mục cơ sở dữ liệu.

Search algorithms can be classified based on their mechanism of searching into 3 types of algorithms: linear, binary, & hashing. Linear search algorithms check every record for the one associated with a target key in a linear fashion. Binary, or half-interval, searches repeatedly target the center of the search structure & divide the search space in half. Comparison search algorithms improve on linear searching by successively eliminating records based on comparisons of the keys until the target record is found, & can be applied on data structures with a defined order. Digital search algorithms work based on the properties of digits in data structures by using numerical keys. Finally, hashing directly maps keys to records based on a hash function.

– Thuật toán tìm kiếm có thể được phân loại dựa trên cơ chế tìm kiếm của chúng thành ba loại thuật toán: tuyến tính, nhị phân, & băm. Thuật toán tìm kiếm tuyến tính kiểm tra mọi bản ghi để tìm bản ghi được liên kết với khóa mục tiêu theo cách tuyến tính. Tìm kiếm nhị phân hoặc nửa khoảng, liên tục nhắm mục tiêu vào tâm của cấu trúc tìm kiếm và chia đôi không gian tìm kiếm. Thuật toán tìm kiếm so sánh cải thiện tìm kiếm tuyến tính bằng cách loại bỏ liên tiếp các bản ghi dựa trên các phép so sánh khóa cho đến khi tìm thấy bản ghi mục tiêu và có thể được áp dụng trên các cấu trúc dữ liệu có thứ tự được xác định. Thuật toán tìm kiếm kỹ thuật số hoạt động dựa trên các thuộc tính của chữ số trong cấu trúc dữ liệu bằng cách sử dụng các khóa số. Cuối cùng, băm trực tiếp ánh xạ khóa thành các bản ghi dựa trên hàm băm.

Algorithms are often evaluated by their computational complexity, or maximum theoretical run time. Binary search functions, e.g., have a maximum complexity of $O(\log n)$, or logarithmic time. In simple terms, the maximum number of operations needed to find the search target is a logarithmic function of the size of the search space.

– Thuật toán thường được đánh giá theo độ phức tạp tính toán hoặc thời gian chạy lý thuyết tối đa. E.g., hàm tìm kiếm nhị phân có độ phức tạp tối đa là $O(\log n)$ hoặc thời gian logarith. Nói 1 cách đơn giản, số lượng thao tác tối đa cần thiết để tìm mục tiêu tìm kiếm là hàm logarit của kích thước không gian tìm kiếm.

1.3.1 Applications of search algorithms – Ứng dụng của thuật toán tìm kiếm

Specific applications of search algorithms include:

- Problems in combinatorial optimization, e.g.:
  - The vehicle routing problem, a form of shortest path problem
  - The knapsack problem: Given a set of items, each with a weight & a value, determine the number of each item to include in a collection so that the total weight is $\leq$ a given limit & the total value is as large as possible.
  - The nurse scheduling problem
- Problems in constraint satisfaction, e.g.:
  - The map coloring problem
  - Filling in a sudoku or crossword puzzle
- In game theory & especially combinatorial game theory, choosing the best move to make next (e.g. with the minmax algorithm)
- Finding a combination or password from the whole set of possibilities
- Factoring an integer (an important problem in cryptography)
- Search engine optimization (SEO) & content optimization for web crawlers
- Optimizing an industrial process, e.g. a chemical reaction, by changing the parameters of the process (like temperature, pressure, & pH)
- Retrieving a record from a database
- Finding the maximum or minimum value in a list or array
- Checking to see if a given value is present in a set of values

1.3.2 Classes of search algorithms

1. **For virtual search spaces.** Algorithms for searching virtual spaces are used in the constraint satisfaction problem, where the goal is to find a set of value assignments to certain varieties that will satisfy specific mathematical equations & inequations/equalities. They are also used when the goal is to find a variable assignment that will maximize or minimize a certain function of those variables. Algorithms for these problems include the basic brute-force search (also called "naïve" or "uninformed" search), & a variety of heuristics that try to exploit partial knowledge about the structure of this space, e.g. linear relaxation, constraint generation, & constraint propagation.

   – Các thuật toán tìm kiếm không gian ảo được sử dụng trong bài toán thỏa mãn ràng buộc, trong đó mục tiêu là tìm 1 tập hợp các phép gán giá trị cho các biến nhất định sẽ thỏa mãn các phương trình và bất phương trình toán học/bằng nhau cụ thể. Chúng cũng được sử dụng khi mục tiêu là tìm 1 phép gán biến sẽ tối đa hóa hoặc tối thiểu hóa 1 hàm nhất định của các biến đó. Các thuật toán cho các bài toán này bao gồm tìm kiếm brute-force cơ bản (còn gọi là tìm kiếm "ngây thơ" hoặc "không có thông tin"), và nhiều phương pháp tìm kiếm khác nhau cố gắng khai thác kiến thức 1 phần về cấu trúc của không gian này, chẳng hạn như thư giãn tuyến tính, tạo ràng buộc và truyền ràng buộc.

   An important subclass are the local search methods, that view the elements of the search space as the vertices of a graph, with edges defined by a set of heuristics applicable to the case; & scan the space by moving from item to item along the edges, e.g. according to the steepest descent or best 1st criterion, or in a stochastic search. This category includes a great variety of general metaheuristic methods, e.g. simulated annealing, tabu search, A-teams, & general programming, that combine arbitrary heuristics in specific ways. The opposite of local search would be global search methods. This method is applicable when the search space is not limited & all aspects of the given network are available to the entity running the search algorithm.

   – 1 phân lớp quan trọng là các phương pháp tìm kiếm cục bộ, xem các phần tử của không gian tìm kiếm như các đỉnh của 1 đồ thị, với các cạnh được xác định bởi 1 tập hợp các phương pháp tìm kiếm áp dụng cho trường hợp này; và quét không gian bằng cách di chuyển từ mục này sang mục khác dọc theo các cạnh, ví dụ theo tiêu chí dốc nhất hoặc tiêu chí tốt nhất đầu tiên, hoặc trong tìm kiếm ngẫu nhiên. Thể loại này bao gồm nhiều phương pháp siêu tìm kiếm chung, chẳng hạn như ủ mô phỏng, tìm kiếm tabu, nhóm A và lập trình di truyền, kết hợp các phương pháp tìm kiếm tùy ý theo những cách cụ thể. Ngược lại với tìm kiếm cục bộ sẽ là các phương pháp tìm kiếm toàn cục. Phương pháp này có thể áp dụng khi không gian tìm kiếm không bị giới hạn và tất cả các khía cạnh của mạng đã cho đều khả dụng đối với thực thể chạy thuật toán tìm kiếm.

   This class also includes various tree search algorithms, that view the elements as vertices of a tree, & traverse that tree in some special order. Examples of the latter include the exhaustive methods e.g. depth-1st search & breadth-1st search, as well as various heuristic-based search tree pruning methods e.g. backtracking & branch & bound. Unlike general metaheuristics, which at best work only in a probabilistic scene, many of these tree-search methods are guaranteed to find the exact or optimal solution, if given enough time. This is called "completeness".

– Lớp này cũng bao gồm nhiều thuật toán tìm kiếm cây khác nhau, xem các phần tử như các đỉnh của 1 cây và duyệt cây đó theo 1 thứ tự đặc biệt nào đó. Ví dụ về thứ tự sau bao gồm các phương pháp đầy đủ như tìm kiếm theo chiều sâu và tìm kiếm theo chiều rộng, cũng như nhiều phương pháp cắt tỉa cây tìm kiếm dựa trên phương pháp heuristic như quay lui và rẽ nhánh và giới hạn. Không giống như các siêu phương pháp heuristic chung, tốt nhất chỉ hoạt động theo nghĩa xác suất, nhiều phương pháp tìm kiếm cây này được đảm bảo tìm ra giải pháp chính xác hoặc tối ưu, nếu có đủ thời gian. Điều này được gọi là "hoàn chỉnh".

Another important subclass consists of algorithms for exploring the game tree of multiple-player games, e.g. chess or backgammon, whose nodes consist of all possible game situations that could result from the current situation. The goal in these problems is to find the move that provides the best chance of a win, taking into account all possible moves of the opponent(s). Similar problems occur when humans or machines have to make successive decisions whose outcomes are not entirely under one's control, e.g. in robot guidance or in marketing, financial, or military strategy planning. This kind of problem – combinatorial search – has been extensively studied in the context of AI. Examples of algorithms for this class are the minimax algorithm, alpha–beta pruning, & the A* algorithm & its variants.

– 1 phân lớp quan trọng khác bao gồm các thuật toán để khám phá cây trò chơi của các trò chơi nhiều người chơi, chẳng hạn như cờ vua hoặc cờ cá ngựa, có các nút bao gồm tất cả các tình huống trò chơi có thể xảy ra do tình huống hiện tại. Mục tiêu của các bài toán này là tìm ra nước đi mang lại cơ hội chiến thắng cao nhất, có tính đến tất cả các nước đi có thể có của đối thủ. Các bài toán tương tự xảy ra khi con người hoặc máy móc phải đưa ra các quyết định liên tiếp mà kết quả không hoàn toàn nằm trong tầm kiểm soát của mình, chẳng hạn như trong hướng dẫn rô-bốt hoặc trong lập kế hoạch chiến lược tiếp thị, tài chính hoặc quân sự. Loại bài toán này – tìm kiếm kết hợp – đã được nghiên cứu rộng rãi trong bối cảnh trí tuệ nhân tạo. Các ví dụ về thuật toán cho lớp này là thuật toán minimax, cắt tỉa alpha–beta và thuật toán A* cùng các biến thể của nó.

2. **For sub-structures of a given structure.** An important & extensively studied subclass are the graph algorithms, in particular graph traversal algorithms, for finding specific sub-structures in a given graph – e.g. subgraphs, paths, circuits, etc. Examples include Dijkstra's algorithm, Kruskal's algorithm, the nearest neighbor algorithm, & Prim's algorithm.

   – *Đối với các cấu trúc con của 1 cấu trúc nhất định.* 1 phân lớp quan trọng và được nghiên cứu rộng rãi là các thuật toán đồ thị, đặc biệt là các thuật toán duyệt đồ thị, để tìm các cấu trúc con cụ thể trong 1 đồ thị nhất định — chẳng hạn như các đồ thị con, đường dẫn, mạch, etc. Các ví dụ bao gồm thuật toán Dijkstra, thuật toán Kruskal, thuật toán hàng xóm gần nhất và thuật toán Prim.

   Another important subclass of this category are the string searching algorithms, that search for patterns within strings. 2 famous examples are the Boyer–Moore & Knuth–Morris–Pratt algorithms, & several algorithms based on the suffix tree data structure.

   – 1 phân lớp quan trọng khác của thể loại này là các thuật toán tìm kiếm chuỗi, tìm kiếm các mẫu trong chuỗi. Hai ví dụ nổi tiếng là các thuật toán Boyer–Moore & Knuth–Morris–Pratt, & 1 số thuật toán dựa trên cấu trúc dữ liệu cây hậu tố.

3. **Search for the maximum of a function.** In 1953, American statistician Jack Kiefer devised Fibonacci search which can be used to find the maximum of a unimodal function & has many other applications in CS.

   – *Tìm kiếm giá trị lớn nhất của 1 hàm số.* Vào năm 1953, nhà thống kê người Mỹ Jack Kiefer đã phát minh ra thuật toán tìm kiếm Fibonacci có thể được sử dụng để tìm giá trị lớn nhất của 1 hàm số đơn thức và có nhiều ứng dụng khác trong khoa học máy tính.

4. **For quantum computers.** There are also search methods designed for quantum computers, like Grover's algorithm, that are theoretically faster than linear or brute-force search even without the help of data structures or heuristics. While the ideas & applications behind quantum computers are still entirely theoretical, studies have been conducted with algorithms like Grover's that accurately replicate the hypothetical physical versions of quantum computing systems.

   – *Đối với máy tính lượng tử:* Cũng có những phương pháp tìm kiếm được thiết kế cho máy tính lượng tử, như thuật toán Grover, về mặt lý thuyết nhanh hơn tìm kiếm tuyến tính hoặc tìm kiếm bằng vũ lực ngay cả khi không có sự trợ giúp của cấu trúc dữ liệu hoặc phương pháp tìm kiếm. Trong khi các ý tưởng và ứng dụng đằng sau máy tính lượng tử vẫn hoàn toàn là lý thuyết, các nghiên cứu đã được tiến hành với các thuật toán như thuật toán Grover sao chép chính xác các phiên bản vật lý giả định của hệ thống máy tính lượng tử.

1.4 Gradient – Độ dốc

Resources – Tài nguyên.

1. [Tiệ25]. Vũ Hữu Tiệp. *Machine Learning Cơ Bản.* Chap. 12: Gradient Descent.

Ví dụ 1 ([Tiệ25], p. 160)**.** *Xét hàm số $f(x) = x^2 + 5\sin x$, $f \in C(\mathbb{R}$ có đạo hàm $f'(x) = 2x + 5\cos x$. Giả sử xuất phát từ 1 điểm x_0, quy tắc cập nhật tại vòng lặp thứ t là*

$$x_{t+1} = x_t - \eta f'(x_t) = x_t - \eta(2x_t + 5\cos x_t).$$

Codes:

- *Python:*

```
import math
import numpy as np

# f(x) = x^2 + 5sin x
def f(x):
    return x**2 + 5*np.sin(x)

def df(x): # derivative f'(x) of f(x)
    return 2*x + 5 * np.cos(x)

x = float(input("x = "))
print("f(x) = ", f(x))
print("df(x) = ", df(x))

tol = 1e-3 # tolerance: just a small number

def gradient_descent(x0, eta): # x0: starting point, eta: learning rate
    x = [x0]
    for i in range(100):
        x_new = x[-1] - eta*df(x[-1]) # x_new: x_{t+1}, x[-1]: x_t
        if abs(df(x_new)) < tol:
            break
        x.append(x_new)
    return(x, i)

x0 = float(input("x0 = "))
eta = float(input("eta = "))
if eta <= 0:
    print("error: eta must be positive!")
else:
    print(gradient_descent(x0, eta))
```

Bài toán 1. *Xét hàm số* $f(x) = x^3 + 3x^2 + 5\sin x - 7\cos x + \sqrt{2}e^{-2x}$. *Viết chương trình* C/C++, Python *để: (a) Tính hàm* $f(x), f'(x)$ *với* $x \in \mathbb{R}$ *được nhập từ bàn phím. (b) Viết hàm gradient descent theo công thức*

$$x_{t+1} = x_t - \eta f'(x_t),$$

với $\eta \in (0, \infty)$ *được gọi là* tốc độ học (learning rate).

Chứng minh. Dễ thấy $f(x)$ là 1 hàm liên tục trên $\mathbb{R}$, i.e., $f \in C(\mathbb{R}$, & có đạo hàm $f'(x) = 3x^2 + 6x + 5\cos x + 7\sin x - 2\sqrt{2}e^{-2x}$.
Code Python:

```
# f1(x) = x^3 + 3x^2  + 5sin x - 7cos x + sqrt{2}e^{-2x}
def f1(x):
    return x**3 + 3*x**2 + 5*np.sin(x) - 7*np.cos(x) + np.sqrt(2)*np.exp(-2*x)

def df1(x):
    return 3*x**2 + 6*x + 5*np.cos(x) + 7*np.sin(x) - 2*np.sqrt(2)*np.exp(-2*x)

x = float(input("x = "))
print("f(x) = ", f(x))
print("df(x) = ", df(x))

tol = 1e-3 # tolerance: just a small number

def gradient_descent_f1(x0, eta): # x0: starting point, eta: learning rate
    x = [x0]
    for i in range(100):
        x_new = x[-1] - eta*df1(x[-1]) # x_new: x_{t+1}, x[-1]: x_t
        if abs(df1(x_new)) < tol:
            break
        x.append(x_new)
    return(x, i)
```

```
x0 = float(input("x0 = "))
eta = float(input("eta = "))
if eta <= 0:
    print("error: eta must be positive!")
else:
    print(gradient_descent_f1(x0, eta))
```

□

Remark 1. *Có thể tham khảo các công thức tính đạo hàm ở Wikipedia/tables of derivatives.*

Bài toán 2. *Xét hàm số* $f(x,y) = 2x^3y^2 + \dfrac{\sqrt{x^3}}{y} + \sin(x^2y) + e^{\cos(xy^2)}$. *Viết chương trình* C/C++, Python *để: (a) Tính hàm* $f(x,y), \nabla f(x,y)$ *với* $x, y \in \mathbb{R}$ *được nhập từ bàn phím. (b) Viết hàm gradient descent cho 2 trường hợp:*

$$(x_{t+1}, y_{t+1}) = (x_t, y_t) - \eta \nabla f(x_t, y_t),$$

or

$$\begin{cases} x_{t+1} = x_t - \boldsymbol{\alpha} \cdot \nabla f(x_t, y_t) = x_t - \alpha_1 \partial_x f(x_t, y_t) - \alpha_2 \partial_x f(x_t, y_t), \\ y_{t+1} = y_t - \boldsymbol{\beta} \cdot \nabla f(x_t, y_t) = x_t - \beta_1 \partial_x f(x_t, y_t) - \beta_2 \partial_x f(x_t, y_t), \end{cases}$$

Python:

```
# f(x,y) = 2x^3y^2 + sqrt(x^3)/y + sin(x^2y) + e^{cos(xy^2)}

def f(x, y):
    return 2*x**3*y**2 + np.sqrt(x**3)/y + np.sin(x**2 * y) + np.exp(np.cos(x * y**2))

def grad_f(x, y):
    df_dx = 6*x**2 * y**2 + (3/2) * x**0.5 / y + 2*x*y * np.cos(x**2 * y) - y**2 * np.sin(x * y**2) * np.exp(np.
    df_dy = 4*x**3 * y - np.sqrt(x**3) / y**2 + x**2 * np.cos(x**2 * y) - 2*x*y * np.sin(x * y**2) * np.exp(np.c
    return np.array([df_dx, df_dy])

x = float(input("x = "))
y = float(input("y = "))
print("f(x,y) = ", f(x,y))
print("grad f(x,y) = ", grad_f(x,y))
```

1.5 Minimax

Resources – Tài nguyên.

1. Wikipedia/minimax.

Minimax (sometimes *Minmax, MM, or saddle point*) is a decision rule used in AI, decision theory, game theory, statistics, & philosophy for *minimizing* the possible loss for a worse case (maximum loss) scenario. When dealing with gains, it is referred to as "maximin" – to maximize the minimum gain. Originally formulated for several-player zero-sum game theory, covering both the cases where players take alternate moves & those where they make simultaneous moves, it has also been extended to more complex games & to general decision-making in the presence of uncertainty.

– *Minimax* (đôi khi là *Minmax, MM*, hoặc *điểm yên ngựa*) là 1 quy tắc quyết định được sử dụng trong trí tuệ nhân tạo, lý thuyết quyết định, lý thuyết trò chơi, thống kê và triết học để giảm thiểu tổn thất có thể xảy ra cho 1 kịch bản xấu nhất (tổn thất tối đa). Khi giải quyết các khoản lợi nhuận, nó được gọi là "maximin" – để tối đa hóa lợi nhuận tối thiểu. Ban đầu được xây dựng cho lý thuyết trò chơi tổng bằng không của nhiều người chơi, bao gồm cả các trường hợp mà người chơi thực hiện các nước đi thay thế và các trường hợp mà họ thực hiện các nước đi đồng thời, nó cũng đã được mở rộng sang các trò chơi phức tạp hơn và ra quyết định chung khi có sự không chắc chắn.

2 Heuristic Algorithms – Các Thuật Giải Heuristic

- **heuristic** [a] (formal) heuristic teaching or education encourages you to learn by discovering things for yourself.
- **heuristics** [n] [uncountable] (formal) a method of solving problems by finding practical ways of dealing with them, learning from past experience.

Resources – Tài nguyên.

1. [**Norvig_Russell2021**]. Peter Norvig, Stuart Russell. *Artificial Intelligence: A Modern Approach*. 4e.
2. Wikipedia/heuristic (CS).

2.1 Heuristic (Computer Science) – Tự tìm tòi (Khoa Học Máy Tính)

In mathematical optimization & CS, *heuristic* (from Greek "I find, discover") is a technique designed for problem solving more quickly when classic methods are too slow for finding an exact or approximate solution, or when classic methods fail to find any exact solution in a search space. This is achieved by trading optimality, completeness, accuracy or precision for speed. In a way, it can be considered a shortcut.

– Trong tối ưu hóa toán học và khoa học máy tính, heuristic (từ tiếng Hy Lạp "Tôi tìm thấy, khám phá") là 1 kỹ thuật được thiết kế để giải quyết vấn đề nhanh hơn khi các phương pháp cổ điển quá chậm để tìm ra giải pháp chính xác hoặc gần đúng, hoặc khi các phương pháp cổ điển không tìm thấy bất kỳ giải pháp chính xác nào trong không gian tìm kiếm. Điều này đạt được bằng cách đánh đổi tính tối ưu, tính hoàn chỉnh, tính chính xác hoặc độ chính xác để lấy tốc độ. Theo 1 cách nào đó, nó có thể được coi là 1 lối tắt.

A *heuristic function*, also simply called a *heuristic*, is a function that ranks alternatives in serch algorithms at each branching step based on available information to decide which brand to follow. E.g., it may approximate the exact solution.

– 1 hàm heuristic, hay còn gọi đơn giản là heuristic, là 1 hàm xếp hạng các phương án thay thế trong thuật toán tìm kiếm tại mỗi bước phân nhánh dựa trên thông tin có sẵn để quyết định nhánh nào sẽ theo. Ví dụ, nó có thể xấp xỉ giải pháp chính xác.

Motivation of heuristic. The objective of a heuristic is to produce a solution in a reasonable time frame that is good enough for solving the problem at hand. This solution may not be the best of all the solutions to this problem, or it may simply approximate the exact solution. But it is still valuable because finding it does not require a prohibitively long time.

– **Động lực của phương pháp tìm kiếm.** Mục tiêu của phương pháp tìm kiếm là đưa ra 1 giải pháp trong 1 khung thời gian hợp lý đủ tốt để giải quyết vấn đề đang xét. Giải pháp này có thể không phải là giải pháp tốt nhất trong tất cả các giải pháp cho vấn đề này hoặc có thể chỉ đơn giản là xấp xỉ giải pháp chính xác. Nhưng nó vẫn có giá trị vì việc tìm ra nó không đòi hỏi quá nhiều thời gian.

Heuristics may produce results by themselves, or they may be used in conjunction with optimization algorithms to improve their efficiency (e.g., they may be used to generate good seed values).

– Phương pháp tìm kiếm có thể tự tạo ra kết quả hoặc có thể được sử dụng kết hợp với các thuật toán tối ưu hóa để cải thiện hiệu quả của chúng (ví dụ: chúng có thể được sử dụng để tạo ra các giá trị hạt giống tốt).

Results about NP-hardness in theoretical computer science make heuristics the only viable option for a variety of complex optimization problems that need to be routinely solved in real-world applications.

– Kết quả về độ khó NP trong khoa học máy tính lý thuyết khiến phương pháp tìm kiếm trở thành lựa chọn khả thi duy nhất cho nhiều vấn đề tối ưu hóa phức tạp cần được giải quyết thường xuyên trong các ứng dụng thực tế.

Heuristics underlie the whole field of AI & the computer simulation of thinking, as they may be used in situations where there are no known algorithms.

– Phương pháp tìm kiếm là nền tảng cho toàn bộ lĩnh vực AI & mô phỏng suy nghĩ bằng máy tính, vì chúng có thể được sử dụng trong những tình huống không có thuật toán nào được biết đến.

Simpler problem. 1 way of achieving the computational performance gain expected of a heuristic consists of solving a simpler problem whose solution is also a solution to the initial problem.

– *Bài toán đơn giản hơn.* 1 cách để đạt được hiệu suất tính toán mong đợi của 1 phương pháp tìm kiếm là giải 1 bài toán đơn giản hơn mà giải pháp của nó cũng là giải pháp cho bài toán ban đầu.

Example 1 (Search)**.** *An example of heuristic making an algorithm faster occurs in certain search problems. Initially, the heuristic tries every possibility at each step, like the full-space search algorithm. But it can stop the search at any time if the current possibility is already worse than the best solution already found. In such search problems, a heuristic can be used to try good choices 1st so that bad paths can be eliminated early, see Wikipedia/alpha-beta pruning. In the case of best-1st search algorithms, e.g. A⋆ search, the heuristic improves the algorithm's convergence while maintaining its correctness as long as the heuristic is admissible.*

– 1 ví dụ khác về phương pháp tìm kiếm giúp thuật toán nhanh hơn xảy ra trong 1 số bài toán tìm kiếm nhất định. Ban đầu, phương pháp tìm kiếm thử mọi khả năng ở mỗi bước, giống như thuật toán tìm kiếm toàn không gian. Nhưng nó có thể dừng tìm kiếm bất kỳ lúc nào nếu khả năng hiện tại đã tệ hơn giải pháp tốt nhất đã tìm thấy. Trong các bài toán tìm kiếm như vậy, phương pháp tìm kiếm có thể được sử dụng để thử các lựa chọn tốt trước để các đường dẫn xấu có thể bị loại bỏ sớm (xem cắt tỉa alpha–beta). Trong trường hợp các thuật toán tìm kiếm tốt nhất trước, chẳng hạn như tìm kiếm A⋆, phương pháp tìm kiếm cải thiện sự hội tụ của thuật toán trong khi vẫn duy trì tính chính xác của nó miễn là phương pháp tìm kiếm được chấp nhận.

Example 2 (Newell & Simon: heuristic search hypothesis)**.** *In their Turing Award acceptance speech, Allen Newell & Herbert A. Simon discuss the heuristic search hypothesis: a physical symbol system will repeatedly generate & modify known symbol structures until the created structure matches the solution structure. Each following step depends upon the step before it, thus the heuristic search learns what avenues to pursue & which ones to disregard by measuring how close the current step is to the solution. Therefore, some possibilities will never be generated as they are measured to be less likely to complete the solution.*

– Trong bài phát biểu nhận giải thưởng Turing, Allen Newell và Herbert A. Simon thảo luận về giả thuyết tìm kiếm theo phương pháp heuristic: 1 hệ thống ký hiệu vật lý sẽ liên tục tạo ra và sửa đổi các cấu trúc ký hiệu đã biết cho đến khi cấu trúc được tạo ra khớp với cấu trúc giải pháp. Mỗi bước tiếp theo phụ thuộc vào bước trước đó, do đó tìm kiếm theo phương pháp heuristic tìm hiểu những con đường nào cần theo đuổi và những con đường nào cần bỏ qua bằng cách đo lường mức độ gần của bước hiện tại với giải pháp. Do đó, 1 số khả năng sẽ không bao giờ được tạo ra vì chúng được đo lường là ít có khả năng hoàn thành giải pháp hơn.

A heuristic method can accomplish its task by using search trees. However, instead of generating all possible solution branches, a heuristic selects branches more likely to produce outcomes than other branches. It is selective at each decision point, picking branches that are more likely to produce solutions.

– 1 phương pháp heuristic có thể hoàn thành nhiệm vụ của mình bằng cách sử dụng cây tìm kiếm. Tuy nhiên, thay vì tạo ra tất cả các nhánh giải pháp khả thi, 1 phương pháp heuristic sẽ chọn các nhánh có nhiều khả năng tạo ra kết quả hơn các nhánh khác. Nó có tính chọn lọc tại mỗi điểm quyết định, chọn các nhánh có nhiều khả năng tạo ra giải pháp hơn.

Example 3 (Antivirus software – Phần mềm diệt virus)**.** *Antivirus software often uses heuristic rules for detecting viruses & other forms of malware. Heuristic scanning looks for code &/or behavioral patterns common to a class or family of viruses, with different sets of rules for different viruses. If a file or executing process is found to contain matching code patterns &/or to be performing that set of activities, then the scanner infers that the file is infected. The most advanced part of behavior–based heuristic scanning is that it can work against highly randomized self-modifying/mutating (polymorphic) viruses that cannot be easily detected by simpler string scanning methods. Heuristic scanning has the potential to detect future viruses without requiring the virus to be 1st detected somewhere else, submitted to the virus scanner developer, analyzed, & a detection update for the scanner provided to the scanner's users.*

– Phần mềm diệt vi-rút thường sử dụng các quy tắc heuristic để phát hiện vi-rút và các dạng phần mềm độc hại khác. Quét heuristic tìm kiếm mã và/hoặc các mẫu hành vi phổ biến đối với 1 lớp hoặc họ vi-rút, với các bộ quy tắc khác nhau cho các loại vi-rút khác nhau. Nếu phát hiện thấy 1 tệp hoặc quy trình thực thi có chứa các mẫu mã trùng khớp và/hoặc đang thực hiện bộ hoạt động đó, thì trình quét sẽ suy ra rằng tệp đó đã bị nhiễm. Phần tiên tiến nhất của quét heuristic dựa trên hành vi là nó có thể hoạt động chống lại các vi-rút tự sửa đổi/đột biến (đa hình) ngẫu nhiên cao mà không thể dễ dàng phát hiện bằng các phương pháp quét chuỗi đơn giản hơn. Quét heuristic có khả năng phát hiện vi-rút trong tương lai mà không cần phải phát hiện vi-rút trước ở nơi khác, gửi cho nhà phát triển trình quét vi-rút, phân tích và cung cấp bản cập nhật phát hiện cho trình quét cho người dùng trình quét.

2.1.1 Pitfalls of heuristic – Những cạm bẫy của phương pháp tìm kiếm

Some heuristics have a strong underlying theory; they are either derived in a top-down manner from the theory or are arrived at based on either experimental or real world data. Others are just rules of thumb based on real-world observation or experience without even a glimpse of theory. The latter are exposed to a larger number of pitfalls.

– 1 số phương pháp tìm kiếm có lý thuyết cơ bản mạnh mẽ; chúng được suy ra theo cách từ trên xuống từ lý thuyết hoặc được đưa ra dựa trên dữ liệu thực nghiệm hoặc dữ liệu thực tế. Những phương pháp khác chỉ là các quy tắc kinh nghiệm dựa trên quan sát hoặc kinh nghiệm thực tế mà không hề có 1 chút lý thuyết nào. Những phương pháp sau có nhiều cạm bẫy hơn.

When a heuristic is reused in various contexts because it has been to "work" in 1 context, without having been mathematically proven to meet a given set of requirements, it is possible that the current data set does not necessarily represent future data sets (see overfitting) & that purported "solutions" turn out to be akin to noise.

– Khi 1 phương pháp tìm kiếm được tái sử dụng trong nhiều bối cảnh khác nhau vì nó được coi là "có hiệu quả" trong 1 bối cảnh, nhưng chưa được chứng minh về mặt toán học là đáp ứng được 1 tập hợp các yêu cầu nhất định, thì có khả năng là tập dữ liệu hiện tại không nhất thiết đại diện cho các tập dữ liệu trong tương lai (xem quá khớp) và các "giải pháp" được cho là giống như nhiễu.

Statistical analysis can be conducted when employing heuristics to estimate the probability of incorrect outcomes. To use a heuristic for solving a search problem or a knapsack problem, it is necessary to check that the heuristic is admissible. Given a heuristic function $h(v_i, v_g)$ meant to approximate the true optimal distance $d^\star(v_i, v_g)$ to the goal node v_g in a directed graph G containing n total nodes or vertices labeled $v_0, v_1, \ldots, v_n$, "admissible" means roughly that the heuristic underestimates the cost to the goal or formally that $h(v_i, v_g) \leq d^\star(v_i, v_g)$, $\forall(v_i, v_g)$ where $i, g \in \{0, 1, \ldots, n\}$.

– Phân tích thống kê có thể được tiến hành khi sử dụng phương pháp tìm kiếm để ước tính xác suất của các kết quả không chính xác. Để sử dụng phương pháp tìm kiếm để giải quyết bài toán tìm kiếm hoặc bài toán ba lô, cần phải kiểm tra xem phương pháp tìm kiếm đó có thể chấp nhận được hay không. Với 1 hàm phương pháp tìm kiếm $h(v_i, v_g)$ có nghĩa là xấp xỉ khoảng cách tối ưu thực sự $d^\star(v_i, v_g)$ đến nút đích v_g trong đồ thị có hướng G chứa tổng cộng n nút hoặc đỉnh được gắn nhãn $v_0, v_1, \ldots, v_n$, "có thể chấp nhận được" có nghĩa là phương pháp tìm kiếm đó ước tính thấp chi phí đến đích hoặc chính thức là $h(v_i, v_g) \leq d^\star(v_i, v_g)$, $\forall(v_i, v_g)$ trong đó $i, g \in \{0, 1, \ldots, n\}$.

If a heuristic is not admissible, it may never find the goal, either by ending up in a dead end of graph G or by skipping back & forth between 2 nodes v_i, v_j where $i, j \neq g$.

– Nếu 1 phương pháp tìm kiếm không được chấp nhận, nó có thể không bao giờ tìm thấy mục tiêu, hoặc là kết thúc ở ngõ cụt của đồ thị G hoặc bằng cách bỏ qua & tiến giữa 2 nút v_i, v_j trong đó $i, j \neq g$.

Some types of heuristics:

1. Constructive heuristic.
2. Metaheuristic: Methods for controlling & tuning basic heuristic algorithms, usually with usage of memory & learning.

   – Siêu thuật toán tìm kiếm: Phương pháp kiểm soát & điều chỉnh các thuật toán tìm kiếm cơ bản, thường sử dụng bộ nhớ & học tập.
3. Matheuristics: Optimization algorithms made by the interoperation of metaheuristics & mathematical programming (MP) techniques.

   – Thuật toán tìm kiếm Toán học: Các thuật toán tối ưu hóa được tạo ra bằng cách kết hợp các kỹ thuật siêu thuật toán và kỹ thuật Toán Tối Ưu (MP).

4. Reactive search optimization: Methods using online ML principles for self-tuning of heuristics.

   – Tối ưu hóa tìm kiếm phản ứng: Phương pháp sử dụng nguyên lý học máy trực tuyến để tự điều chỉnh phương pháp tìm kiếm.

2.2 Admissible heuristics – Các phương pháp tìm kiếm có thể chấp nhận được

Resources – Tài nguyên.

1. Wikipedia/admissible heuristic.

In CS, specifically in algorithms related to pathfinding, a heuristic function is said to be *admissible* if it never overestimates the cost of reaching the goal, i.e., the cost it estimates to reach the goal is not higher than the lowest possible cost from the current point in the path. I.e., it should act as a lower bound.

– Trong khoa học máy tính, đặc biệt là trong các thuật toán liên quan đến tìm đường, 1 hàm heuristic được cho là có thể chấp nhận được nếu nó không bao giờ ước tính quá cao chi phí để đạt được mục tiêu, tức là chi phí mà nó ước tính để đạt được mục tiêu không cao hơn chi phí thấp nhất có thể từ điểm hiện tại trên đường đi. Nói cách khác, nó sẽ hoạt động như 1 giới hạn dưới.

It is related to the concept of consistent heuristics. While all consistent heuristics are admissible, not all admissible heuristics are consistent.

– Nó liên quan đến khái niệm về phương pháp tìm kiếm nhất quán. Trong khi tất cả các phương pháp tìm kiếm nhất quán đều có thể chấp nhận được, không phải tất cả các phương pháp tìm kiếm được chấp nhận đều nhất quán.

$$\text{consistent heuristic} \not\Longleftrightarrow \text{admissible heuristic}$$

Search algorithms. An admissible heuristic is used to estimate the cost of reaching the goal state in an informed search algorithm. In order for a heuristic to be admissible to the search problem, the estimated cost must always be $\leq$ the actual cost of reaching the goal state. The search algorithm uses the admissible heuristic to find an estimated optimal path to the goal state from the current node. E.g., in A* search the evaluation function, where n is the current node, is $f(n) = g(n) + h(n)$, where $f(n)$: the evaluation function, $g(n)$: the cost from the start node to the current node, & $h(n)$: estimated cost from current node to goal. $h(n)$ is calculated using the heuristic function. With a non-admissible heuristic, the A* algorithm could overlook the optimal solution to a search problem due to an overestimation in $f(n)$.

– *Thuật toán tìm kiếm.* 1 phương pháp tìm kiếm có thể chấp nhận được được sử dụng để ước tính chi phí đạt được trạng thái mục tiêu trong 1 thuật toán tìm kiếm có thông tin. Để 1 phương pháp tìm kiếm có thể chấp nhận được đối với bài toán tìm kiếm, chi phí ước tính phải luôn bằng $\leq$ chi phí thực tế để đạt được trạng thái mục tiêu. Thuật toán tìm kiếm sử dụng phương pháp tìm kiếm có thể chấp nhận được để tìm đường dẫn tối ưu ước tính đến trạng thái mục tiêu từ nút hiện tại. Ví dụ, trong tìm kiếm A*, hàm đánh giá, trong đó n là nút hiện tại, là $f(n) = g(n) + h(n)$, trong đó $f(n)$: hàm đánh giá, $g(n)$: chi phí từ nút bắt đầu đến nút hiện tại, & $h(n)$: chi phí ước tính từ nút hiện tại đến mục tiêu. $h(n)$ được tính toán bằng cách sử dụng hàm phương pháp tìm kiếm. Với 1 phương pháp tìm kiếm không được chấp nhận, thuật toán A* có thể bỏ qua giải pháp tối ưu cho 1 bài toán tìm kiếm do ước tính quá cao trong $f(n)$.

Formulation. n is a node, h is a heuristic, $h(n)$ is cost indicated by h to reach a goal from n, $h^\star(n)$ is the optimal cost to reach a goal from n. Then $h(n)$ is *admissible* if $h(n) \leq h^\star(n)$, $\forall$ node n.

Construction. An admissible heuristic can be derived from a relaxed version of the problem, or by information from pattern databases that store exact solutions to subproblems of the problem, or by using inductive learning methods.

– 1 phương pháp tìm kiếm có thể chấp nhận được có thể được rút ra từ phiên bản đơn giản của bài toán, hoặc từ thông tin từ cơ sở dữ liệu mẫu lưu trữ các giải pháp chính xác cho các bài toán con của bài toán, hoặc bằng cách sử dụng các phương pháp học quy nạp.

2.2.1 Optimality proof of admissible heuristics

If an admissible heuristic is used in an algorithm that, per iteration, progresses only the path of lowest evaluation (current cost + heuristic) of several candidate paths, terminates the moment its exploration reaches the goal &, crucially, never closes all optimal paths before terminating (something that's possible with A* search algorithm if special case isn't taken), then this algorithm can only terminate on an optimal path. To see why, consider the following proof by contradiction:

– Nếu 1 phương pháp tìm kiếm có thể chấp nhận được được sử dụng trong 1 thuật toán, theo mỗi lần lặp, chỉ tiến triển theo đường đánh giá thấp nhất (phương pháp tìm kiếm chi phí hiện tại +) của 1 số đường ứng viên, kết thúc ngay khi quá trình khám phá của nó đạt đến mục tiêu &, quan trọng là không bao giờ đóng tất cả các đường tối ưu trước khi kết thúc (điều này có thể thực hiện được với thuật toán tìm kiếm A* nếu không áp dụng trường hợp đặc biệt), thì thuật toán này chỉ có thể kết thúc trên 1 đường tối ưu. Để biết lý do, hãy xem xét bằng chứng phản chứng sau:

Assume such an algorithm managed to terminate on a path T with a true cost T_{true} greater than the optimal path S with true cost S_{true}. I.e., before terminating, the evaluated cost of T was $\leq$ the evaluated cost of S (or else S would have been picked). Denote these evaluated costs $T_{\text{eval}}, S_{\text{eval}}$, resp. The above can be summarized as follows,

$$S_{\text{true}} < T_{\text{true}},\ T_{\text{eval}} \leq S_{\text{eval}}.$$

If our heuristic is admissible it follows that at this penultimate step $T_{\text{eval}} = T_{\text{true}}$ because any increase on the true cost by the heuristic on T would be inadmissible & the heuristic cannot be negative. On the other hand, an admissible heuristic would

require that $S_{\text{eval}} \leq S_{\text{true}}$ which combined with the above inequalities gives us $T_{\text{eval}} < T_{\text{true}}$ & more specifically $T_{\text{eval}} \neq T_{\text{true}}$. As $T_{\text{eval}}, T_{\text{true}}$ cannot be both equal & unequal our assumption must have been false & so it must be impossible to terminate on a more costly than optimal path.

– Giả sử 1 thuật toán như vậy đã kết thúc trên 1 đường dẫn T với chi phí thực T_{true} lớn hơn đường dẫn tối ưu S với chi phí thực S_{true}. Tức là, trước khi kết thúc, chi phí được đánh giá của T bằng $\leq$ chi phí được đánh giá của S (nếu không thì S đã được chọn). Ký hiệu các chi phí được đánh giá này là $T_{\text{eval}}, S_{\text{eval}}$, tương ứng. Có thể tóm tắt những điều trên như sau,

$$S_{\text{true}} < T_{\text{true}},\ T_{\text{eval}} \leq S_{\text{eval}}.$$

Nếu phương pháp heuristic của chúng ta có thể chấp nhận được thì theo đó tại bước áp chót này $T_{\text{eval}} = T_{\text{true}}$ vì bất kỳ sự gia tăng nào về chi phí thực tế của phương pháp heuristic trên T đều không thể chấp nhận được & phương pháp heuristic không thể là số âm. Mặt khác, 1 phương pháp heuristic có thể chấp nhận được sẽ yêu cầu $S_{\text{eval}} \leq S_{\text{true}}$ kết hợp với các bất đẳng thức trên cho chúng ta $T_{\text{eval}} < T_{\text{true}}$ & cụ thể hơn là $T_{\text{eval}} \neq T_{\text{true}}$. Vì $T_{\text{eval}}, T_{\text{true}}$ không thể vừa bằng nhau & không bằng nhau nên giả định của chúng ta phải sai & do đó không thể kết thúc trên 1 đường dẫn tốn kém hơn đường dẫn tối ưu.

Although an admissible heuristic can guarantee final optimality, it is not necessarily efficient.

– Mặc dù 1 phương pháp tìm kiếm có thể chấp nhận được có thể đảm bảo tính tối ưu cuối cùng, nhưng nó không nhất thiết phải hiệu quả.

2.3 Consistent heuristic

Resources – Tài nguyên.

1. Wikipedia/consistent heuristic.

In the study of path-finding problems in AI, a heuristic function is said to be *consistent*, or *monotone*, if its estimate is always $\leq$ the estimated distance from any neighboring vertex to the goal, plus the cost of reaching that neighbor.

Formally, for every node N & each successor P of N, the estimated cost of reaching the goal from N is $\leq$ the step cost of getting to P plus the estimated cost of reaching the goal from P. I.e.:

$$h(N) \leq c(N, P) + h(P),\ h(G) = 0,$$

where

- h is the consistent heuristic function
- N is any node in the graph
- $P$is any descendant of N

3 Scheduling problem – Bài toán phân công

Problem 1. *What can you learn about scheduling problem with* C/C++, Python *implementation via Google &/or current best AIs.*

Dạng toán 1. *Cài đặt & đánh giá thực nghiệm 1 thuật giải heuristic cho bài toán phân công công việc (đơn giản), & thuật giải cải tiến.*

Bài toán 3 (Roster – Bài toán phân công đơn giản)**.** *1 đề án gồm $n \in \mathbb{N}^\star$ công việc & các việc sẽ được thực hiện bởi $m \in \mathbb{N}^\star$ máy như nhau. Giả sử biết thời gian để 1 máy thực hiện việc thứ i là t_i. Yêu cầu: Tìm phương án phân công sao cho thời gian hoàn thành toàn bộ công việc là thấp nhất.*

Input. *m: số máy, n: số việc, dãy $t[0], \ldots, t[n-1]$ với $t[i]$: thời gian để 1 máy thực hiện việc i.*

Output. *Bảng phân công tối ưu.*

Sample.

scheduling.inp	scheduling.out
3 10 4 9 5 2 7 6 10 8 7 5	

Thuật giải cho bài toán phân công đơn giản – Pseudocode.

Mathematical analyse – Phân tích Toán học. Gọi n công việc là $w_1, \ldots, w_n$ (w: work), gọi m máy là $M_1, \ldots, M_m$ (các máy này có công suất làm việc như nhau). Yêu cầu của bài toán: Phân hoạch tập $\{t_i\}_{i=1}^n$ thành m tập con $T_1, \ldots, T_m$ lần lượt có số phần tử là $n_1, \ldots, n_m$, i.e., $|T_i| = n_i$, $\forall i = 1, \ldots, m$. □

Computer Science analyse – Phân tích Tin học. □

```
for (i = 0; i < n; i++) {
    chọn việc i chưa phân công có thời gian thực hiện cao nhất;
    chọn máy m có thời gian làm việc thấp nhất;
    bố trí việc i cho máy m;
}
```

- Input: https://github.com/NQBH/advanced_STEM_beyond/blob/main/AI/Python/scheduling.inp.
- Output: https://github.com/NQBH/advanced_STEM_beyond/blob/main/AI/Python/scheduling.out.
- Python: https://github.com/NQBH/advanced_STEM_beyond/blob/main/AI/Python/scheduling.py.

```
m, n = map(int, input().split())
t = [int(x) for x in input().split()]
d = [0] * m # devices/machines's current accomplished time
t.sort(reverse = True) # descending order
for i in range(n):
    current_max_work = t[0] # current longest work
    t.pop(0) # remove current longest work
    # print(t)
    d.sort() # ascending order
    # print(d)
    d[0] += int(current_max_work) # laziest device takes longest work
    # print(d)
print(max(d))
```

Các bước `print` để mô phỏng quá trình giao công việc cho các máy để tiện hình dung, không bắt buộc.

Bài toán 4 (Extended scheduling – Bài toán phân công mở rộng)**.** *Có $n \in \mathbb{N}^\star$ công việc & $m \in \mathbb{N}^\star$ máy không đồng nhất. Biết thời gian máy i làm việc j là $t_{ij} = t[i][j]$. Yêu cầu: Lập bảng phân công tối ưu.*

Input. *m: số máy, n: số việc, array 2 chiều $t[i][j]$: thời gian để máy i thực hiện việc j.*

Output. *Bảng phân công tối ưu.*

Sample.

extended_scheduling.inp	extended_scheduling.out
3 8 4 5 4 10 8 6 12 8 7 5 7 3 9 7 9 5 10 6 7 12 10 6 5 7	

Cách phát biểu khác của bài toán phân công mở rộng. Có $n \in \mathbb{N}^\star$ công việc sẽ được phân công cho $m \in \mathbb{N}^\star$ người thực hiện, mỗi việc được phân công cho 1 người. Giả sử ta biết thời gian $t_{ij} = t[i][j]$ cần để người thứ i thực hiện công việc thứ j, $\forall i = 1, \dots, m$, $\forall j = 1, \dots, n$. Tìm 1 phương pháp phân công sao cho thời gian hoàn thành tất cả các công việc là thấp nhất.

Mathematics analysis. Bài toán phân công công việc mở rộng này có thể được mô tả toán học như sau: Có $n \in \mathbb{N}^\star$ máy (variable: `number_machine`). Phân hoạch tập hợp $[n] = \{1, 2, \dots, n\}$ thành m tập con (có thể rỗng, i.e., máy tương ứng làm biếng, không nhận bất cứ không việc nào cả[1]) $M_1, M_2, \dots, M_m$:

$$M_j = \{i_{j,1}, i_{j,2}, \dots, i_{j,n_i}\},\ \forall j =$$

[INSERTING+++]

3.1 Optimal job scheduling – Lên lịch công việc tối ưu

Resources – Tài nguyên.

1. Wikipedia/optimal job scheduling.

Optimal job scheduling is a class of optimization problem related to scheduling. The inputs to such problems are a list of jobs (also called *processes* or *tasks*) & a list of machines (also called *processors* or *workers*). The required output is a *schedule* – an assignment of jobs to machines. The schedule should optimize a certain objective function. In the literature, problems of optimal job scheduling are often called *machine scheduling, processor scheduling, multiprocessor scheduling*, or just *scheduling*.

– Lên lịch công việc tối ưu là 1 lớp các vấn đề tối ưu hóa liên quan đến việc lên lịch. Đầu vào cho các vấn đề như vậy là 1 danh sách các công việc (còn gọi là quy trình hoặc tác vụ) và 1 danh sách các máy (còn gọi là bộ xử lý hoặc công nhân). Đầu ra bắt buộc là 1 lịch trình – 1 sự phân công các công việc cho các máy. Lịch trình này phải tối ưu hóa 1 hàm mục tiêu nhất định.

[1] Nếu đổi máy thành người, nhân viên thì nên sa thải hoặc bị xã hội đào thải.

Trong tài liệu, các vấn đề về lập lịch công việc tối ưu thường được gọi là lập lịch máy, lập lịch bộ xử lý, lập lịch đa bộ xử lý hoặc chỉ là lập lịch.

There are many different problems of optimal job scheduling, different in the nature of jobs, the nature of machines, the restrictions on the schedule, & the objective function. A convenient *notation* for optimal scheduling problems was introduced by Ronald Graham, Eugene Lawler, Jan Karel Lenstra, & Alexander Rinnooy Kan. It consists of 3 fields: α, β, γ. Each field may be a comma separated list of words. The α field describes the machine environment, β the job characteristics & constraints, & γ the objective function. Since its introduction in the late 1970s the notation has been constantly extended, sometimes inconsistently. As a result, today there are some problems that appear with distinct notations in several papers.

– Có rất nhiều bài toán khác nhau về lập kế hoạch công việc tối ưu, khác nhau về tính chất công việc, tính chất của máy móc, những hạn chế về tiến độ, & hàm mục tiêu. 1 *notation* thuận tiện cho các vấn đề lập kế hoạch tối ưu đã được giới thiệu bởi Ronald Graham, Eugene Lawler, Jan Karel Lenstra, & Alexander Rinnooy Kan. Nó bao gồm 3 trường: α, β, γ. Mỗi trường có thể là 1 danh sách các từ được phân tách bằng dấu phẩy. Trường α mô tả môi trường máy, β mô tả đặc điểm công việc & ràng buộc, & γ mô tả hàm mục tiêu. Kể từ khi được giới thiệu vào cuối những năm 1970, ký hiệu này đã liên tục được mở rộng, đôi khi không nhất quán. Do đó, hiện nay có 1 số vấn đề xuất hiện với các ký hiệu riêng biệt trong 1 số bài báo.

3.1.1 Single-stage jobs vs. multi-stage jobs – Công việc 1 giai đoạn so với công việc nhiều giai đoạn

In the simpler optimal job scheduling problems, each job j consists of a single execution phase, with a given processing time p_j. In more complex variants, each job consists of several execution phases, which may be executed in sequence or in parallel.

– Trong các bài toán lập lịch công việc tối ưu đơn giản hơn, mỗi công việc j bao gồm 1 giai đoạn thực hiện duy nhất, với thời gian xử lý nhất định p_j. Trong các biến thể phức tạp hơn, mỗi công việc bao gồm 1 số giai đoạn thực hiện, có thể được thực hiện theo trình tự hoặc song song.

3.1.2 Machine environments – Môi trường máy móc

In *single-stage job scheduling problems*, there are 4 main categories of machine environments:

1. **1**: single-machine scheduling. There is a single machine.
2. **P**: identical-machines scheduling. There are $m \in \mathbb{N}$, $m \geq 2$, parallel machines, & they are identical. Job j takes time $p_j \in (0, \infty)$ on any machine it is scheduled to.
3. **Q**: uniform-machines scheduling. There are $m \in \mathbb{N}$, $m \geq 2$, parallel machines, & they have different given speeds. Job j on machine i takes time $\frac{p_j}{s_i}$.
4. **R**: unrelated-machines scheduling. There are m parallel machines, & they are unrelated – job j on machine i takes time p_{ij}.

These letters might be followed by the number of machines, which is then fixed. E.g., **P2** indicates that there are 2 parallel identical machines. **Pm** indicates that there are $m \in \mathbb{N}$, $m \geq 2$, parallel identical machines, where m is a fixed parameter. In contrast, **P** indicates that there are m parallel identical machines, but m is not fixed (it is part of the input).

In *multi-stage job scheduling problems*, there are other options of the machine environments:

1. **O**: open-shop problem. Every job j consists of m operations O_{ij} for $i \in [m]$. The operations can be scheduled in *any* order. Operation O_{ij} must be processed for p_{ij} units on machine i.
2. **F**: Flow-shop problem. Every job j consists of m operations O_{ij} for $i \in [m]$, to be scheduled in the given order. Operation O_{ij} must be processed for p_{ij} units on machine i.
3. **J**: job-shop problem. Every job j consists of n_j operations O_{kj} for $k \in [n_j]$, to be scheduled in that order. Operation O_{kj} must be processed for p_{kj} units on a *dedicated* machine μ_{kj} with $\mu_{kj} \neq \mu_{k'j}$ for $k \neq k'$.

3.1.3 Job characteristics – Đặc điểm công việc

All processing times are assumed to be integers. In some odler research papers however they are assumed to be rationals.

1. $p_i = p$, or $p_{ij} = p$: the processing time is equal for all jobs.
2. $p_i = 1$, or $p_{ij} = 1$: the processing time is equal to 1 time-unit for all jobs.
3. r_j: for each job a release time is given before which it cannot be scheduled, default is 0.
4. online-r_j: an online problem. Jobs are revealed at their release times. In this context the performance of an algorithm is measured by its competitive ratio.
5. d_j: for each job a due date is given. The idea is that every job should complete before its due date & there is some penalty for jobs that complete late. This penalty is denoted in the objective value. The presence of the job characteristic d_j is implicitly assumed & not denoted in the problem name, unless there are some restrictions as e.g. $d_j = d$, assuming that all due dates are equal to some given date.
6. $\bar{d}_j$: for each job a strict deadline is given. Every job must complete before its deadline.

7. `pmtn`: Jobs can be preempted & resumed possibly on another machine. Sometimes also denoted by `prmp`/

8. size$_j$: each job comes with a number of machines on which it must be scheduled at the same time. The default is 1. This is an important parameter in the variant called parallel task scheduling.

Precedence relations. Each pair of 2 jobs may or may not have a precedence relation. A precedence relation between 2 jobs means that 1 job must be finished before the other job. E.g., if job i is a predecessor of job j in that order, job j can only start once job i is completed.

- `prec`: There are no restrictions placed on the precedence relations.
- `chains`: Each job is the predecessor of at most 1 other job & is preceded by at most 1 other job.
- `tree`: The precedence relations must satisfy 1 of the 2 restrictions.
  1. `intree`: Each node is the predecessor of at most 1 other job.
  2. `outtree`: Each node is preceded by at most 1 other job.
- `opposing forest`: If the graph of precedence relations is split into connected components, then each connected component is either an intree or outtree.
- `sp-graph`: The graph of precedence relation is a series parallel graph.
- `bounded height`: The length of the longest directed path is capped at a fixed value. (A directed path is a sequence of jobs where each job except the last is a predecessor of the next job in the sequence.)
- `level order`: Each job has a level, which is the length of the longest directed path starting from that job. Each job with level k is a predecessor of every job with level $k-1$.
- `interval order`: Each job x has an interval $[s_x, e_x)$ & job x is a predecessor of y iff the end of the interval of x is strictly less than the start of the interval for y.

In the presence of a precedence relation one might in addition assume *time lags*. The time lag between 2 jobs is the amount of time that must be waited after 1st job is complete before the 2nd job to begin. Formally, if job i precedes job j, then $C_i + l_{ij} \leq S_j$ must be true. If no time lag l_{ij} is specified then it is assumed to be 0. Time lags can also be negative. A negative time lag means that the 2nd job can begin a fixed time before the 1st job finishes.

1. l: The time lag is the same for each pair of jobs.
2. l_{ij}: Different pairs of jobs can have different time lags.

Transportation delays.

- t_{jk}: Between the completion of operation O_{kj} of job j on machine k & the start of operation $O_{k+1,j}$ of job j on machine $k+1$, there is a transportation delay of at least t_{jk} units.
- t_k: Machine dependent transportation delay. Between the completion of operation O_{kj} of job j on machine k & the start of operation $O_{k+1,j}$ of job j on machine $k+1$, there is a transportation delay of at least t_k units.
- t_{kl}: Machine pair dependent transportation delay. Between the completion of operation O_{kj} of job j on machine k & the start of operation $O_{l,j}$ of job j on machine l, there is a transportation delay of at least t_{kl} units.
- t_j: Job dependent transportation delay. Between the completion of operation O_{kj} of job j on machine k & the start of operation $O_{l,j}$ of job j on machine l, there is a transportation delay of at least t_j units.

Various constraints.

- `rcrc`: Also known as Recirculation or flexible job shop. The promise on μ is lifted & for some pairs $k \neq k'$ we might have $\mu_{kj} = \mu_{k'j}$. I.e., it is possible for different operations of the same job to be assigned to the same machine.
- `no-wait`: The operation $O_{k+1,i}$ must start exactly when operation $O_{k,i}$ completes. I.e., once 1 operation of a job finishes, the next operation must begin immediately. Sometimes also denoted as `nwt`.
- `no-idle`: No machine may ever be idle between the start of its 1st execution to the end of its last execution.
- size$_j$: Multiprocessor tasks on identical parallel machines. The execution of job j is done simultaneously on size$_j$ parallel machines.
- fix$_j$: Multiprocessor tasks. Every job j is given with a set of machines fix$_j \subset [m]$, & needs simultaneously all these machines for execution. Sometimes also denoted by 'MPT'.
- M_j: Multipurpose machines. Every job j needs to be scheduled on 1 machine out of a given set $M_j \subset [m]$. Sometimes also denoted by M_j.

3.1.4 Objective functions – Các hàm mục tiêu

Usually the goal is to minimize some objective value. 1 difference is the notation $\sum_j U_j$ where the goal is to maximize the number of jobs that complete before their deadline. This is also called the *throughput*. The objective value can be sum, possibly weighted by some given priority weights w_j per job.

- -: The absence of an objective value is denoted by a single dash. I.e., the problem consists simply in producing a feasible scheduling, satisfying all given constraints.
- C_j: the *completion time* of job j. $C_{\max}$: maximum completion time; also known as the makespan. Sometimes we are interested in the *mean* completion time (the average of C_j over all j), which is sometimes denoted by `mft` (mean finish time).
- F_j: The *flow time* of a job is the difference between its completion time & its release time, i.e., $F_j = C_j - r_j$.
- L_j: Lateness. Every job j is given a due date d_j. The lateness of job j is defined as $C_j - d_j$. Sometimes $L_{\max}$ is used to denote feasibility for a problem with deadlines. Indeed using binary search, the complexity of the feasibility version is equivalent to the minimization of $L_{\max}$.
- U_j: Throughout. Every job is given a due date d_j. There is a unit profit for jobs that complete on time, i.e., $U_j = 1$ if $C_j \leq d_j$ & $U_j = 0$ otherwise. Sometimes the meaning of U_j is inverted in the literature, which is equivalent when considering the decision version of the problem, but which makes a huge difference for approximations.
- T_j: Tardiness. Every job j is given a due date d_j. The tardiness of job j is defined as $T_j = \max\{0, C_j - d_j\} = (C_j - d_j)_+$.
- E_j: Earliness. Every job j is given a due date d_j. The earliness of job j is defined as $E_j = \max\{0, d_j - C_j\} = (d_j - C_j)_+$. This objective is important for *just-in-time scheduling*.

There are also variants with multiple objectives, but they are much less studied.

Here are some examples for problems defined using the above notation.

- $P_2 \parallel C_{\max}$: assigning each of n given jobs to 1 of the 2 identical machines so to minimize the maximum total processing time over the machines. This is an optimization version of the partition problem.
- $1|\text{prec}|L_{\max}$: assigning to a single machine, processes with general precedence constraint, minimizing maximum lateness.
- $\text{R}|\text{pmtn}|\sum C_i$: assigning tasks to a variable number of unrelated parallel machines, allowing preemption, minimizing total completion time.
- $\text{J3}|p_{ij} = 1|C_{\max}$: a 3-machine job shop problem with unit processing times, where the goal is to minimize the maximum completion time.
- $P|\text{size}_j|C_{\max}$: assigning jobs to m parallel identical machines, where each job comes with a number of machines on which it must be scheduled at the same time, minimizing maximum completion time. See parallel task scheduling.

Other variants of optimal job scheduling:

- All variants surveyed above are *deterministic* in that all data is known to the planner. There are also *stochastic* variants, in which the data is not known in advance, or can perturb randomly.
- In a *load balancing game*, each job belongs to a strategic agent, who can decide where to schedule his job. The Nash equilibrium in this game may not be optimal. Aumann & Dombb assess the inefficiency of equilibrium in several load-balancing games.

3.2 Fractional job scheduling – Lập lịch công việc theo từng phần

Resources – Tài nguyên.

1. Wikipedia/fractional job scheduling.

Fractional job scheduling is a variant of optimal job scheduling in which it is allowed to break jobs into parts & process each part separately on the same or a different machine. Breaking jobs into parts may allow for improving the overall performance, e.g., decreasing the makespan. Moreover, the computational problem of finding an optimal schedule may become easier, as some of the optimization variables become continuous. On the other hand, breaking jobs apart might be costly.

– Lập lịch công việc theo phần là 1 biến thể của lập lịch công việc tối ưu trong đó cho phép chia nhỏ công việc thành nhiều phần và xử lý từng phần riêng biệt trên cùng 1 máy hoặc 1 máy khác. Chia nhỏ công việc thành nhiều phần có thể cho phép cải thiện hiệu suất chung, ví dụ, giảm thời gian hoàn thành. Hơn nữa, vấn đề tính toán để tìm lịch trình tối ưu có thể trở nên dễ dàng hơn, vì 1 số biến tối ưu hóa trở nên liên tục. Mặt khác, việc chia nhỏ công việc có thể tốn kém.

There are several variants of job scheduling problems in which it is allowed to break jobs apart. They can be broadly classified into *quyền ưu tiên & tách, chia nhỏ ra*.

1. In the preemption variants, different parts of a job must be processed at different times. In the 3-field notation, they are denoted by *pmtn*. They were 1st studied by McNaughton.

2. In the splitting variants, different parts of a job may be processed simultaneously on different machines. They are denoted by *split* & were introduced by Xing & Zhang.

– Có 1 số biến thể của các vấn đề lập lịch công việc trong đó cho phép chia nhỏ các công việc. Chúng có thể được phân loại thành *preemption & splitting*.

1. Trong các biến thể preemption, các phần khác nhau của 1 công việc phải được xử lý tại các thời điểm khác nhau. Trong ký hiệu 3 trường, chúng được ký hiệu là *pmtn*. Chúng được nghiên cứu lần đầu tiên bởi McNaughton.

2. Trong các biến thể chia nhỏ, các phần khác nhau của 1 công việc có thể được xử lý đồng thời trên các máy khác nhau. Chúng được ký hiệu là *split* & được giới thiệu bởi Xing & Zhang.

For more details on fractional job scheduling, see, e.g., Wikipedia/fractional job scheduling.

4 Bài toán tô màu đồ thị – Graph coloring problem

Bài toán 5 (Bài toán tô màu các đỉnh đồ thị – Graph coloring problem)**.** *Có 1 đồ thị vô hướng đơn giản. Ta muốn tìm cách tô màu cho các đỉnh của đồ thị sao cho 2 đỉnh cạnh nhau phải có màu khác nhau. Yêu cầu: Tìm phương án tô sao cho số màu sử dụng là ít nhất.*

Input. *Đồ thị vô hướng đơn giản.*

Output. *Mỗi đỉnh tô màu gì.*

1 thuật giải heuristic. Sử dụng nguyên lý thứ tự:

```
for (i = 0; i < n; i++) {
    chọn đỉnh s chưa tô có d[s] lớn nhất;
    chọn màu: ưu tiên tô đỉnh s bằng 1 trong các màu đã sử dụng, nếu không được thì sử dụng màu mới;
    sau khi tô màu cho đỉnh s: với mỗi đỉnh x cạnh, giảm d[x]; ???
}
```

$d[x]$: số đỉnh cạnh x mà chưa tô màu. ???

4.1 Shortest path problem – Bài toán đường đi ngắn nhất

Bài toán 6. *Cài đặt & thử nghiệm A*. So sánh với Dijkstra nếu được.*

Input. *$G = (V, E)$ có trọng số dương, đỉnh xuất phát a, đỉnh mục tiêu z. Thông tin bổ sung: $h(x)$: ước lượng khoảng cách từ a đến mục tiêu z.*

Output. *Đường đi ngắn nhất shortest path SP từ a đến z.*

4.2 Traveling salesman problem (TSP) – Bài toán người bán hàng du lịch

Resources – Tài nguyên.

1. Wikipedia/traveling salesman problem.

An example of approximation is described by Jon Bentley for solving the traveling salesman problem (TSP):

Problem 2 (Original Traveling Salesman Problem (TSP))**.** *Given a list of cities & the distances between each pair of cities, what is the shortest possible route that visits each city exactly once & returns to the origin city?*

– Cho 1 danh sách các thành phố & khoảng cách giữa mỗi cặp thành phố, đâu là tuyến đường ngắn nhất có thể đi qua mỗi thành phố đúng 1 lần & quay trở lại thành phố ban đầu?

so as to select the order to draw using a pen plotter. TSP is known to be NP-hard so an optimal solution for even a moderate size problem is difficult to solve. Instead, the greedy algorithm can be used to give a good but not optimal solution (it is an approximation to the optimal answer) in a reasonably short amount of time. The greedy algorithm heuristic says to pick whatever is currently the best next step regardless of whether that prevents (or even makes impossible) good steps later. It is a heuristic in the sense that practice indicates it is a good enough solution, while theory indicates that there are better solutions (& even indicates how much better, in some cases).

– Jon Bentley đã đưa ra 1 ví dụ về phép xấp xỉ để giải bài toán người bán hàng du lịch (TSP): để chọn thứ tự vẽ bằng bút vẽ. TSP được biết là NP-khó nên giải pháp tối ưu cho ngay cả bài toán có kích thước vừa phải cũng khó giải. Thay vào đó, thuật toán tham lam có thể được sử dụng để đưa ra giải pháp tốt nhưng không tối ưu (là phép xấp xỉ với câu trả lời tối ưu) trong 1 khoảng thời gian khá ngắn. Thuật toán tham lam nói rằng hãy chọn bất kỳ bước tiếp theo nào hiện là tốt nhất bất kể điều đó có ngăn cản (hoặc thậm chí khiến không thể) thực hiện các bước tốt sau này hay không. Đây là thuật toán theo nghĩa là thực hành chỉ ra rằng đó là giải pháp đủ tốt, trong khi lý thuyết chỉ ra rằng có những giải pháp tốt hơn (& thậm chí chỉ ra tốt hơn bao nhiêu, trong 1 số trường hợp).

Problem 3 (Traveling Salesman Problem (TSP)). *The traveling salesman must visit every city in this territory exactly once & then return to the starting point; given the cost of travel between all cities, how should he plan his itinerary for minimum total cost of the entire tour?*

TSP $\in$ NP-Complete.

Remark 2 (Approximate TSP by GAs). *We shall discuss a single possible approach to approximate the TSP by genetic algorithms (GAs).*

4.3 15 Puzzle Problem

Resources – Tài nguyên.

1. Wikipedia/admissible heuristic.

2. Wikipedia/15 puzzle.

2 different examples of admissible heuristics apply to the Wikipedia/15 puzzle problem:

- Hamming distance
- Manhattan distance

The Hamming distance is the total number of misplaced tiles. It is clear that this heuristic is admissible since the total number of moves to order the tiles correctly is at least the number of misplaced tiles (each tile not in place must be moved at least once). The cost (number of moves) to the goal (an ordered puzzle) is at least the Hamming distance of the puzzle.

– Khoảng cách Hamming là tổng số ô bị đặt sai vị trí. Rõ ràng là phương pháp tìm kiếm này có thể chấp nhận được vì tổng số lần di chuyển để sắp xếp các ô đúng ít nhất bằng số ô bị đặt sai vị trí (mỗi ô không đúng vị trí phải được di chuyển ít nhất 1 lần). Chi phí (số lần di chuyển) đến đích (một câu đố có thứ tự) ít nhất bằng khoảng cách Hamming của câu đố.

The Manhattan distance of a puzzle is defined as:

$$h(n) = \sum_{\text{all tiles}} \text{distance}(\text{tile}, \text{correct position}).$$

Consider the puzzle below in which the player wishes to move each tile s.t. the numbers are ordered. The Manhattan distance is an admissible heuristic in this case because every tile will have to be moved at least the number of spots in between itself & its correct position. z – Xem xét câu đố bên dưới trong đó người chơi muốn di chuyển từng ô theo thứ tự các số. Khoảng cách Manhattan là 1 phép thử có thể chấp nhận được trong trường hợp này vì mỗi ô sẽ phải được di chuyển ít nhất số điểm giữa nó & vị trí chính xác của nó.

Bài toán 7 (Tính khoảng cách Hamming & khoảng cách Manhattan). *(a) Cho 1 dãy hoán vị của* [15] *theo thứ tự sắp xếp trái sang phải, trên xuống dưới. Tính khoảng cách Hamming & khoảng cách Manhattan của hoán vị này.*

Input. *Dòng 1 chứa số bộ test* $t \in \mathbb{N}^\star$. *Với* t *dòng tiếp theo, mỗi dòng chứa đúng 1 hoán vị* $\{a_n\}_{n=1}^{15} = a_1, a_2, \ldots, a_{15}$ *của* [15].

Output. *Khoảng cách Hamming & khoảng cách Manhattan của hoán vị* $\{a_n\}_{n=1}^{15}$.

Sample.

15_puzzle.inp	15_puzzle.out
1 4 6 3 8 7 12 9 14 15 13 1 5 2 10 11	$h(n) = 36$

(b) Mở rộng bài toán từ 15 *thành* $n \in \mathbb{N}^\star$.

5 A* Search Algorithm – Thuật Toán Tìm Kiếm A*

Resources – Tài nguyên.

1. Wikipedia/A* search algorithm.

2. Geeks4Geeks/A* search algorithm.

A* (pronounced "A-star") is a graph traversal & pathfinding algorithm that is used in many fields of CS due to its completeness, optimality, & optimal efficiency. Given a weighted graph, a source node & a goal node, the algorithm finds the shortest path (w.r.t. the given weights) from source to goal.

– A* (phát âm là "A-star") là 1 thuật toán duyệt đồ thị & tìm đường được sử dụng trong nhiều lĩnh vực của CS do tính hoàn chỉnh, tối ưu, & hiệu quả tối ưu của nó. Với 1 đồ thị có trọng số, 1 nút nguồn & 1 nút đích, thuật toán tìm đường đi ngắn nhất (so với các trọng số đã cho) từ nguồn đến đích.

1 major practical drawback is its $O(b^d)$ space complexity where d is the depth of the shallowest solution (the length of the shortest path from the source node to any given goal node) & b is the branching factor (the maximum number of successors for any given state), as it stores all generated nodes in memory. Thus, in practical travel-routing systems, it is generally outperformed by algorithms that can pre-process the graph to attain better performance, as well as by memory-bounded approaches; however, A* is still the best solution in many cases.

– 1 nhược điểm thực tế lớn là độ phức tạp không gian $O(b^d)$ trong đó d là độ sâu của giải pháp nông nhất (độ dài của đường đi ngắn nhất từ nút nguồn đến bất kỳ nút đích nào) & b là hệ số phân nhánh (số lượng nút kế nhiệm tối đa cho bất kỳ trạng thái nào), vì nó lưu trữ tất cả các nút được tạo trong bộ nhớ. Do đó, trong các hệ thống định tuyến du lịch thực tế, nó thường bị các thuật toán có thể xử lý trước đồ thị để đạt được hiệu suất tốt hơn, cũng như các phương pháp tiếp cận bị giới hạn bộ nhớ đánh bại; tuy nhiên, A* vẫn là giải pháp tốt nhất trong nhiều trường hợp.

Peter hart, Nils Nilsson, & Bertram Raphael of Stanford Research Institute (now SRI International) 1st published the algorithm in 1968. It can be seen as an extension of Dijkstra's algorithm. A* achieves better performance by using heuristics to guide its search.

– Peter Hart, Nils Nilsson, & Bertram Raphael của Viện nghiên cứu Stanford (nay là SRI International) lần đầu tiên công bố thuật toán này vào năm 1968. Thuật toán này có thể được coi là phần mở rộng của thuật toán Dijkstra. A* đạt được hiệu suất tốt hơn bằng cách sử dụng phương pháp tìm kiếm để hướng dẫn tìm kiếm của nó.

Compared to Dijkstra's algorithm, the A* algorithm only finds the shortest path from a specified source to a specified goal, & not the shortest-path tree from a specified source to all possible goals. This is a necessary trade-off for using a specific-goal-directed heuristics. For Dijkstra's algorithm, since the entire shortest-path tree is generated, every node is a goal, & there can be no specific-goal-directed heuristic.

– So với thuật toán Dijkstra, thuật toán A* chỉ tìm đường đi ngắn nhất từ 1 nguồn cụ thể đến 1 mục tiêu cụ thể, & không phải là cây đường đi ngắn nhất từ 1 nguồn cụ thể đến tất cả các mục tiêu có thể. Đây là 1 sự đánh đổi cần thiết để sử dụng thuật toán tìm kiếm theo mục tiêu cụ thể. Đối với thuật toán Dijkstra, vì toàn bộ cây đường đi ngắn nhất được tạo ra, nên mọi nút đều là 1 mục tiêu, & không thể có thuật toán tìm kiếm theo mục tiêu cụ thể.

Motivation. To approximate the shortest path in real-life situations, like in maps, games where there can be many hindrances. We can consider a 2D Grid having several obstacles & we start from a source cell to reach towards a goal cell.

– Để ước lượng đường đi ngắn nhất trong các tình huống thực tế, như trong bản đồ, trò chơi, nơi có thể có nhiều chướng ngại vật. Chúng ta có thể xem xét 1 Lưới 2D có nhiều chướng ngại vật & chúng ta bắt đầu từ 1 ô nguồn để tiếp cận đến 1 ô đích.

What is A* search algorithm? A* search algorithm is 1 of the best & popular technique used in path-finding & graph traversals.

– Thuật toán tìm kiếm A* là 1 trong những kỹ thuật & phổ biến nhất được sử dụng trong tìm đường & duyệt đồ thị.

Why A* search algorithm? Informally speaking, A* search algorithms, unlike other traversal techniques, it has "brains". I.e., it is really a smart algorithm which separates it from the other conventional algorithms. Many games & web-based map use this algorithm to find the shortest path very efficiently (approximation).

– Nói 1 cách không chính thức, thuật toán tìm kiếm A*, không giống như các kỹ thuật duyệt khác, nó có "bộ não". Tức là, nó thực sự là 1 thuật toán thông minh giúp nó tách biệt với các thuật toán thông thường khác. Nhiều trò chơi & bản đồ dựa trên web sử dụng thuật toán này để tìm đường đi ngắn nhất rất hiệu quả (xấp xỉ).

Explanation. Consider a square grid having many obstacles & we are given a starting cell & a target cell. We want to reach the target cell (if possible) from the starting cell as quickly as possible. Here A* search algorithm comes to the rescue.

– Xem xét 1 lưới vuông có nhiều chướng ngại vật & chúng ta được cung cấp 1 ô bắt đầu & 1 ô đích. Chúng ta muốn tiếp cận ô đích (nếu có thể) từ ô bắt đầu càng nhanh càng tốt. Ở đây, thuật toán tìm kiếm A* sẽ giải cứu.

What A* search algorithm does it that at each step it picks the node according to a value f which is a parameter equal to the sum of 2 other parameters g, h. At each step it picks the node/cell having the lowest f, & process that node/cell. We define g, h as simply as possible:

- g = the movement cost to move from the starting point to a given square on the grid, following the path generated to get there.
- h = the estimated movement cost to move from that given square on the grid to the final destination. This is often referred to as the *heuristic*, which is nothing but a ind of smart guess.

$$\text{heuristics} = \text{smart guess.}$$

We really don't know the actual distance until we find the path, because all sorts of things can be in the ways (walls, water, etc.). There can be many ways to calculate this h discussed later.

– Thuật toán tìm kiếm A* thực hiện điều gì khi ở mỗi bước, nó chọn nút theo giá trị f là tham số bằng tổng của 2 tham số khác g, h. Ở mỗi bước, nó chọn nút/ô có f thấp nhất, & xử lý nút/ô đó. Chúng tôi định nghĩa g, h đơn giản nhất có thể:

- g = chi phí di chuyển để di chuyển từ điểm bắt đầu đến 1 ô vuông cho trước trên lưới, theo đường dẫn được tạo ra để đến đó.
- h = chi phí di chuyển ước tính để di chuyển từ ô vuông cho trước đó trên lưới đến đích cuối cùng. Điều này thường được gọi là *heuristic*, không gì khác ngoài 1 ind của phỏng đoán thông minh.

$$\text{heuristics} = \text{phỏng đoán thông minh.}$$

Chúng ta thực sự không biết khoảng cách thực tế cho đến khi tìm thấy đường đi, vì có thể có nhiều thứ cản đường (tường, nước, v.v.). Có thể có nhiều cách để tính h này được thảo luận sau.

Algorithm. We create 2 lists - open list & closed list (just like Dijkstra algorithm):
A* Search Algorithm.

1. Initialize the open list.
2. Initialize the closed list. Put the startiong node on the open list (you can leave its f at 0).
3. While the open list is not empty:
   (a) Find the node with the least f on the open list, call it q.
   (b) Pop q off the open list.
   (c) Generate q's 8 (why 8?) successors & set their parents to q.
   (d) For each successor
      i. if successor is the goal, stop search.
      ii. else, compute both g, h for successor.
      successor.g = $q.g$ + distance between successor & q
      successor.h = distance from goal to successor (this can be done using many ways, e.g., 3 heuristics: Manhattan, diagonal, & Euclidean heuristics).
      successor.f = successor.g + successor.h
      iii. if a node with the same position as successor is in the OPEN list which has a lower f than successor, skip this successor.
      iv. if a node with the same position as successor is in the CLOSED list which has a lower f than successor, skip this successor otherwise, add the node to the open list
   end (for loop)
   (e) Push q on the closed list.
end (while loop)

5.1 Description of A* algorithm – Mô tả thuật toán A*

A* is an informed search algorithm, or a best-1st search, i.e., it is formulated in terms of weighted graphs: starting from a specific starting node of a graph, it aims to find a path to the given goal node having the smallest cost (least distance traveled, shortest time, etc.). It does this by maintaining a tree of paths originating at the start node & extending those paths 1 edge at a time until the goal node is reached.

– A* là 1 thuật toán tìm kiếm có thông tin, hay tìm kiếm best-1st, tức là nó được xây dựng theo các đồ thị có trọng số: bắt đầu từ 1 nút bắt đầu cụ thể của đồ thị, nó nhằm mục đích tìm 1 đường dẫn đến nút đích đã cho có chi phí nhỏ nhất (khoảng cách di chuyển ngắn nhất, thời gian ngắn nhất, v.v.). Nó thực hiện điều này bằng cách duy trì 1 cây các đường dẫn bắt đầu từ nút bắt đầu & mở rộng các đường dẫn đó thêm 1 cạnh tại 1 thời điểm cho đến khi đạt đến nút đích.

At each iteration of its main loop, A* needs to determine which of its paths to extend. It does so based on the cost of the path & an estimate of the cost required to extend the path all the way to the goal. Specifically, A* selects the path that minimizes $f(n) = g(n) + h(n)$ where n is the next node on the path, $g(n)$ is the cost of the path from the start node to n, & $h(n)$ is a heuristic function that estimates the cost of the cheapest path from n to the goal. The heuristic function is problem-specific. If the heuristic function is admissible, i.e., it never overestimates the actual cost to get to the goal – A* is guaranteed to return a least-cost path from start to goal.

– Tại mỗi lần lặp của vòng lặp chính, A* cần xác định đường dẫn nào của nó sẽ được mở rộng. Nó thực hiện việc này dựa trên chi phí của đường dẫn & ước tính chi phí cần thiết để mở rộng đường dẫn cho đến đích. Cụ thể, A* chọn đường dẫn tối thiểu hóa $f(n) = g(n) + h(n)$ trong đó n là nút tiếp theo trên đường dẫn, $g(n)$ là chi phí của đường dẫn từ nút bắt đầu đến n, & $h(n)$ là hàm heuristic ước tính chi phí của đường dẫn rẻ nhất từ n đến đích. Hàm heuristic là hàm cụ thể cho từng vấn đề. Nếu hàm heuristic có thể chấp nhận được, tức là nó không bao giờ ước tính quá cao chi phí thực tế để đến đích – A* được đảm bảo trả về đường dẫn có chi phí thấp nhất từ điểm bắt đầu đến đích.

Typical implementations of A* use a priority queue to perform the repeated selection of minimum (estimated) cost nodes to expand. This priority queue is known as the *open* set, *fringe* or *frontier*. At each step of the algorithm, the node with the lowest $f(x)$ value is removed from the queue, the f, g values of its neighbors are updated accordingly, & these neighbors are added to the queue. The algorithm continues until a removed node (thus the node with the lowest f value out of all fringe nodes) is a goal node. The f value of that goal is then also the cost of the shortest path, since h at the goal is 0 in an admissible heuristic.

– Các triển khai điển hình của A* sử dụng hàng đợi ưu tiên để thực hiện lựa chọn lặp lại các nút có chi phí tối thiểu (ước tính) để mở rộng. Hàng đợi ưu tiên này được gọi là tập *open*, *fringe* hoặc *frontier*. Tại mỗi bước của thuật toán, nút có giá trị $f(x)$ thấp nhất sẽ bị xóa khỏi hàng đợi, các giá trị f, g của các nút lân cận của nó sẽ được cập nhật tương ứng, & các nút lân cận này sẽ được thêm vào hàng đợi. Thuật toán tiếp tục cho đến khi 1 nút bị xóa (do đó là nút có giá trị f thấp nhất trong số tất cả các nút rìa) là 1 nút mục tiêu. Giá trị f của mục tiêu đó sau đó cũng là chi phí của đường dẫn ngắn nhất, vì h tại mục tiêu là 0 trong 1 thuật toán tìm kiếm có thể chấp nhận được.

The algorithm described so far only gives the length of the shortest path. To find the actual sequence of steps, the algorithm can be easily revised so that each node on the path keeps track of its predecessor. After this algorithm is run, the ending node will point to its predecessor, & so on, until some node's predecessor is the start node.

– Thuật toán được mô tả cho đến nay chỉ cung cấp độ dài của đường đi ngắn nhất. Để tìm trình tự các bước thực tế, thuật toán có thể dễ dàng được sửa đổi để mỗi nút trên đường đi theo dõi nút tiền nhiệm của nó. Sau khi thuật toán này được chạy, nút kết thúc sẽ trỏ đến nút tiền nhiệm của nó, & cứ thế, cho đến khi nút tiền nhiệm của 1 số nút là nút bắt đầu.

E.g., when searching for the shortest route on a map, $h(x)$ might represent the straight-line distance to the goal, since that is physically the smallest possible distance between any 2 points. For a grid map from a video game, using the Taxicab distance or the Chebyshev distance becomes better depending on the set of movements available (4-way or 8-way).

– E.g., khi tìm kiếm tuyến đường ngắn nhất trên bản đồ, $h(x)$ có thể biểu diễn khoảng cách theo đường thẳng đến đích, vì về mặt vật lý, đó là khoảng cách nhỏ nhất có thể giữa bất kỳ 2 điểm nào. Đối với bản đồ lưới từ trò chơi điện tử, sử dụng khoảng cách Taxicab hoặc khoảng cách Chebyshev sẽ tốt hơn tùy thuộc vào tập hợp các chuyển động khả dụng (4 chiều hoặc 8 chiều).

If the heuristic h satisfies the additional condition $h(x) \leq d(x, y) + h(y)$ for every edge (x, y) of the graph (where d denotes the length of that edge), then h is called monotone or consistent. With a consistent heuristic, A* is guaranteed to find an optimal path without processing any node more than once & A* is equivalent to running Dijkstra's algorithm with the reduced cost $d'(x, y) := d(x, y) + h(y) - h(x)$.

– Nếu thuật toán heuristic h thỏa mãn điều kiện bổ sung $h(x) \leq d(x, y) + h(y)$ cho mọi cạnh (x, y) của đồ thị (trong đó d biểu thị độ dài của cạnh đó), thì h được gọi là đơn điệu hoặc nhất quán. Với thuật toán heuristic nhất quán, A* được đảm bảo tìm ra đường đi tối ưu mà không cần xử lý bất kỳ nút nào nhiều hơn 1 lần & A* tương đương với việc chạy thuật toán Dijkstra với chi phí giảm $d'(x, y) := d(x, y) + h(y) - h(x)$.

5.1.1 Pseudocode of A* algorithm – Mã giả của thuật toán A*

The following pseudocode describes the algorithm:

```
function reconstruct_path(cameFrom, current)
    total_path := {current}
    while current in cameFrom.Keys:
        current := cameFrom[current]
        total_path.prepend(current)
    return total_path

// A* finds a path from start to goal.
// h: heuristic function. h(n) estimates the cost to reach goal from node n.
function A_star(start, goal, h)
    // set of discovered nodes that may need to be (re-)expanded.
    // initially, only the start node is known.
    // this is usually implemented as a min-heap or priority queue rather than a hash-set.
    openSet := {start}

    // for node n, cameFrom[n] is the node immediately preceding it on the cheapest path from the start
    cameFrom := an empty map

    // for node n, gScore[n] is the currently known cost of the cheapest path from start to n.
    gScore := map with default value of Infinity
    gScore[start] := 0

    // for node n, fScore[n] := gScore[n] + h(n). fScore[n] represents our current best guess as to\\
    // how cheap a path could be from start to finish if it goes through n.
    fScore := map with default value of Infinity
    fScore[start] := h(start)

    while openSet is not empty
        // this operation can occur in O(log N) time if openSet is a min-heap or a priority queue
        current := the node in openSet having the lowest fScore[] value
        if current = goal
            return reconstruct_path(cameFrom, current)

        openSet.remove(current)
        for each neighbor of current
            // d(current, neighbor) is the weight of the edge from current to neighbor
            // tentative_gScore is the distance from start to the neighbor through current
            tentative_gScore := gScore[current] + d(current, neighbor)
            if tentative_gScore < gScore[neighbor]
```

```
                // this path to neighbor is better than any previous one. Record it!
                cameFrom[neighbor] := current
                gScore[neighbor] := tentative_gScore
                fScore[neighbor] := tentative_gScore + h(neighbor)
                if neighbor not in openSet
                    openSet.add(neighbor)
    // open set is empty but goal was never reached
    return failure
```

Remark 3. *In this pseudocode, if a node is reached by* 1 *path, removed from* `openSet`*, & subsequently reached by a cheaper path, it will be added to openSet again. This is essential to guarantee that the path returned is optimal if the heuristic function is admissible but not consistent. If the heuristic is consistent, when a node is removed from* `openSet` *the path to it is guaranteed to be optimal so the test* `tentative_gScore < gScore[neighbor]` *will always fail if the node is reached again. The pseudocode implemented here is sometimes called the* graph-search *version of A*. This is in contrast with the version without* `tentative_gScore < gScore[neighbor]` *test to add nodes back to* `openSet`*, which is sometimes called the* tree-search *version of A* & require a consistent heuristic to guarantee optimality.*

– Trong mã giả này, nếu 1 nút được tiếp cận bằng 1 *đường dẫn, được xóa khỏi* `openSet`*, & sau đó được tiếp cận bằng 1 đường dẫn rẻ hơn, thì nút đó sẽ được thêm vào openSet 1 lần nữa. Điều này rất cần thiết để đảm bảo rằng đường dẫn được trả về là tối ưu nếu hàm heuristic được chấp nhận nhưng không nhất quán. Nếu heuristic là nhất quán, khi 1 nút được xóa khỏi* `openSet` *thì đường dẫn đến nút đó được đảm bảo là tối ưu, do đó, thử nghiệm* `tentative_gScore < gScore[neighbor]` *sẽ luôn thất bại nếu nút được tiếp cận lại. Mã giả được triển khai ở đây đôi khi được gọi là phiên bản* graph-search *của A*. Điều này trái ngược với phiên bản không có thử nghiệm* `tentative_gScore < gScore[neighbor]` *để thêm các nút trở lại* `openSet`*, đôi khi được gọi là phiên bản* tree-search *của A* & yêu cầu 1 heuristic nhất quán để đảm bảo tính tối ưu.*

Example 4. *An example of an A* algorithm in action where nodes are cities connected with roads & $h(x)$ is the straight-line distance to the target point.*

The A* algorithm has real-world applications.

Example 5. *Edges are railroads & $h(x)$ is the great-circle distance (the shortest possible distance on a sphere) to the target. The algorithm is searching for a path between Washington, D.C., & Los Angeles.*

Example 6. *Illustration of A* search for finding path from a start node to a goal node in a robot motion planning problem. The empty circles represent the nodes in the* open *set, i.e., those that remain to be explored, & the filled ones are in the closed set. Color on each closed node indicates the distance from the goal: the greener, the closer. One can 1st see the A* moving in a straight line in the direction of the goal, then when hitting the obstacle, it explores alternative routes through the nodes from the open set.*

5.1.2 Implementation details of A* search algorithm

5.1.3 Special cases

5.2 Properties of A* search algorithm

5.3 Bounded relaxation

5.4 Complexity of A* search algorithm

5.5 Applications

5.6 Relations to other algorithms

What sets A* apart from a greedy best-1st search algorithm is that it takes the cost/distance already traveled, $g(n)$, into account.

Some common variants of Dijkstra's algorithm can be viewed as a special case of A* where the heuristic $h(n) = 0$ for all nodes; in turn, both Dijkstra & A* are special cases of DP. A* itself is a special case of a generalization of branch & bound.

A* is similar to beam search except that beam search maintains a limit on the numbers of paths that it has to explore.

5.7 Variants of A* search algorithm – Các biến thể của thuật toán tìm kiếm A*

Anytime A*, Block A*, D*, Field D*, Fringe, Fringe Saving A* (FSA*), Generalized Adaptive A* (GAA*), incremental heuristic search, reduced A*, Iterative deepening A* (IDA*), jump point search, Lifelong Planning A* (LPA*), New Bidirectional A* (NBA*), Simplified Memory bounded A* (SMA*), Theta*.

A* can also be adapted to a bidirectional search algorithm, but special care needs to be taken for the stopping criterion.

6 Traveling Salesman Problem (TSP) – Bài Toán Người Bán Hàng Du Lịch

Resources – Tài nguyên.

1. Wikipedia/traveling salesman problem.

Example 7. *The TSP seeks to find the shortest possible loop that connects every red dot.*

In the theory of computational complexity, the *traveling salesman problem* (TSP) asks the following question: "Given a list of cities & the distances between each pair of cities, what is the shortest possible route that visits each city exactly once & returns to the origin city?" It is an NP-hard problem in combinatorial optimization, important in theoretical CS & operations research.

– Trong lý thuyết về độ phức tạp tính toán, bài toán người bán hàng du lịch (TSP) đặt ra câu hỏi sau: "Cho 1 danh sách các thành phố và khoảng cách giữa mỗi cặp thành phố, đâu là tuyến đường ngắn nhất có thể đến thăm mỗi thành phố đúng 1 lần và quay trở lại thành phố ban đầu?" Đây là 1 bài toán NP-khó trong tối ưu hóa tổ hợp, quan trọng trong khoa học máy tính lý thuyết và nghiên cứu hoạt động (vận trù học).

The traveling purchaser problem, the vehicle routing problem, & the ring star problem are 3 generalizations of TSP.

– Vấn đề người mua du lịch, vấn đề định tuyến phương tiện, & vấn đề vòng tròn sao là 3 tổng quát hóa của TSP.

The decision version of the TSP (where given a length L, the task is to decide whether the graph has a tour whose length is at most L) belongs to the class of NP-complete problems. Thus, it is possible that the worst-case running time for any algorithm for the TSP increases superpolynomially (but no more than exponentially) with the number of cities.

– Phiên bản quyết định của TSP (với độ dài L, nhiệm vụ là quyết định xem đồ thị có 1 chuyến tham quan có độ dài tối đa là L hay không) thuộc về lớp các bài toán NP-complete. Do đó, có thể thời gian chạy trường hợp xấu nhất cho bất kỳ thuật toán nào đối với TSP tăng theo siêu đa thức (nhưng không quá cấp số nhân) với số lượng thành phố.

The problem was 1st formulated in 1930 & is 1 of the most intensively studied problems in optimization. It is used as a benchmark for many optimization methods. Even though the problem is computationally difficult, many heuristics & exact algorithms are known, so that some instances with 10s of thousands of cities can be solved completely, & even problems with millions of cities can be approximated within a small fraction of 1%.

– Bài toán này được xây dựng lần đầu tiên vào năm 1930 và là 1 trong những bài toán được nghiên cứu chuyên sâu nhất trong tối ưu hóa. Bài toán này được sử dụng làm chuẩn mực cho nhiều phương pháp tối ưu hóa. Mặc dù bài toán này khó tính toán, nhưng người ta đã biết nhiều phương pháp tìm kiếm và thuật toán chính xác, do đó 1 số trường hợp với hàng chục nghìn thành phố có thể được giải quyết hoàn toàn, và thậm chí các bài toán với hàng triệu thành phố cũng có thể được xấp xỉ trong phạm vi 1 phần nhỏ của 1%.

The TSP has several applications even in its purest formulation, e.g. planning, logistics, & the manufacture of microchips. Slightly modified, it appears as a sub-problem in many cases, e.g. DNA sequencing. In these applications, the concept *city* represents, e.g., customers, soldering points, or DNA fragments, & the concept *distance* represents traveling times or cost, or a similarity measure between DNA fragments. The TSP also appears in astronomy, as astronomers observing many sources wants to minimize the time spent moving the telescope between the sources; in such problems, the TSP can be embedded inside an optimal control problem. In many applications, additional constraints e.g. limited resources or time windows may be imposed.

– TSP có 1 số ứng dụng ngay cả trong công thức tinh khiết nhất của nó, ví dụ như lập kế hoạch, hậu cần, & sản xuất vi mạch. Được sửa đổi 1 chút, nó xuất hiện như 1 bài toán phụ trong nhiều trường hợp, ví dụ như giải trình tự DNA. Trong các ứng dụng này, khái niệm *city* biểu diễn, ví dụ như khách hàng, điểm hàn hoặc đoạn DNA, & khái niệm *distance* biểu diễn thời gian di chuyển hoặc chi phí hoặc thước đo độ tương đồng giữa các đoạn DNA. TSP cũng xuất hiện trong thiên văn học, vì các nhà thiên văn học quan sát nhiều nguồn muốn giảm thiểu thời gian di chuyển kính thiên văn giữa các nguồn; trong các bài toán như vậy, TSP có thể được nhúng bên trong 1 bài toán điều khiển tối ưu. Trong nhiều ứng dụng, các ràng buộc bổ sung ví dụ như tài nguyên hạn chế hoặc cửa sổ thời gian có thể được áp dụng.

6.1 History of TSP – Lịch sử của bài toán người bán hàng du lịch

6.2 Description of TSP – Mô tả bài toán người bán hàng du lịch

6.2.1 Description of TSP as a graph problem

TSP can be modeled as an undirected weighted graph, s.t. cities are the graph's vertices, paths are the graph's edges, & a path's distance is the edge's weight. It is a minimization problem starting & finishing at a specified vertex after having visited each other vertex exactly once. Often, the model is a complete graph (i.e., each pair of vertices is connected by an edge). If no path exists between 2 cities, then adding a sufficiently long edge will complete the graph without affecting the optimal tour.

– TSP có thể được mô hình hóa như 1 đồ thị có trọng số không định hướng, s.t. cities là các đỉnh của đồ thị, paths là các cạnh của đồ thị, & khoảng cách của 1 path là trọng số của cạnh. Đây là 1 bài toán tối thiểu hóa bắt đầu & kết thúc tại 1 đỉnh đã chỉ định sau khi đã đi qua từng đỉnh khác đúng 1 lần. Thông thường, mô hình là 1 đồ thị hoàn chỉnh (tức là mỗi cặp đỉnh được kết nối bằng 1 cạnh). Nếu không có đường đi nào tồn tại giữa 2 thành phố, thì việc thêm 1 cạnh đủ dài sẽ hoàn thành đồ thị mà không ảnh hưởng đến hành trình tối ưu.

6.2.2 Asymmetric & symmetric

In the *symmetric TSP*, the distance between 2 cities is the same in each opposite direction, forming an undirected graph. This symmetry halves the number of possible solutions. In the *asymmetric TSP*, paths may not exist in both directions or the distances might be different, forming a directed graph. Traffic congestion, 1-way streets, & airfares for cities with different departure & arrival fees are real-world considerations that could yield a TSP problem in asymmetric form.

– Trong TSP đối xứng, khoảng cách giữa hai thành phố là như nhau theo mỗi hướng ngược nhau, tạo thành 1 đồ thị vô hướng. Sự đối xứng này làm giảm 1 nửa số lượng các giải pháp khả thi. Trong TSP không đối xứng, các đường đi có thể không

tồn tại theo cả hai hướng hoặc khoảng cách có thể khác nhau, tạo thành 1 đồ thị có hướng. Tình trạng tắc nghẽn giao thông, đường 1 chiều và giá vé máy bay cho các thành phố có mức phí khởi hành và đến khác nhau là những cân nhắc thực tế có thể tạo ra 1 bài toán TSP ở dạng không đối xứng.

6.2.3 Related problems to TSP

1. An equivalent formulation in terms of graph theory is: Given a complete weighted graph (where the vertices would represent the cities, the edges would represent the roads, & the weights would be the cost or distance of that road), find a Hamiltonian cycle with the least weight. This is more general than the Hamiltonian path problem, which only asks if a Hamiltonian path (or cycle) exists in a non-complete unweighted graph.

   – 1 công thức tương đương về mặt lý thuyết đồ thị là: Cho 1 đồ thị có trọng số hoàn chỉnh (trong đó các đỉnh sẽ biểu diễn các thành phố, các cạnh sẽ biểu diễn các con đường, & trọng số sẽ là chi phí hoặc khoảng cách của con đường đó), hãy tìm 1 chu trình Hamilton có trọng số nhỏ nhất. Điều này tổng quát hơn bài toán đường đi Hamilton, chỉ yêu cầu xem có tồn tại đường đi Hamilton (hoặc chu trình) trong 1 đồ thị không có trọng số không hoàn chỉnh hay không.

2. The requirement of returning to the starting city does not change the computational complexity of the problem; see Hamiltonian path problem.

   – Yêu cầu quay trở lại thành phố xuất phát không làm thay đổi độ phức tạp tính toán của bài toán; xem Bài toán đường đi Hamilton.

3. Another related problem is the bottleneck traveling salesman problem: Find a Hamiltonian cycle in a weighted graph with the minimal weightof the weightiest edge.

6.3 Integer linear programming formulations – Công thức lập trình tuyến tính số nguyên

The TSP can be formulated as an integer linear program. Several formulations are known. 2 notable formulations are the Miller-Tucker-Zemlin (MTZ) formulation & the Dantzig–Fulkerson–Johnson (DFJ) formulation. The DFJ formulation is stronger, though the MTZ formulation is still useful in certain settings.

– TSP có thể được xây dựng như 1 chương trình tuyến tính số nguyên. Có 1 số công thức được biết đến. 2 công thức đáng chú ý là công thức Miller-Tucker-Zemlin (MTZ) & công thức Dantzig–Fulkerson–Johnson (DFJ). Công thức DFJ mạnh hơn, mặc dù công thức MTZ vẫn hữu ích trong 1 số cài đặt nhất định.

Common to both these formulations is that one labels the cities with the numbers $1, 2, \ldots, n$ & takes $c_{ij} > 0$ to be the cost (distance) from city i to city j. The main variables in the formulations are

$$x_{ij} = \begin{cases} 1 & \text{the path goes from city } i \text{ to city } j, \\ 0 & \text{otherwise.} \end{cases}$$

It is because these are `0/1` variables that the formulations become integer programs; all other constraints are purely linear. In particular, the objective in the program is to minimize the tour length

$$\sum_{i=1}^{n} \sum_{j \neq i,\, j=1}^{n} c_{ij} x_{ij}.$$

Without further constraints, the $\{x_{ij}\}_{i,j}$ will effectively range over all subsets of the set of edges, which is very far from the sets of edges in a tour & allows for a trivial minimum where all $x_{ij} = 0$. Therefore, both formulations also have the constraints that, at each vertex, there is exactly 1 incoming edge & 1 outgoing edge, which may be expressed as the $2n$ linear equations

$$\begin{cases} \displaystyle\sum_{i=1,\, i \neq j}^{n} x_{ij} = 1, \ \forall j \in [n], \\ \displaystyle\sum_{j=1,\, j \neq i}^{n} x_{ij} = 1, \ \forall i \in [n]. \end{cases}$$

These ensure that the chosen set of edges locally looks like that of a tour, but still allow for solutions violating the global requirement that there is *1* tour which visits all vertices, as the edges chosen could make up several tours, each visiting only a subset of the vertices; arguably, it is this global requirement that makes TSP a hard problem. The MTZ & DFJ formulations differ in how they express this final requirement as linear constraints.

– Điểm chung của cả hai công thức này là người ta dán nhãn các thành phố bằng các số $1, 2, \ldots, n$ & lấy $c_{ij} > 0$ là chi phí (khoảng cách) từ thành phố i đến thành phố j. Các biến chính trong các công thức là

$$x_{ij} = \begin{cases} 1 & \text{đường đi từ thành phố } i \text{ đến thành phố } j, \\ 0 & \text{nếu không.} \end{cases}$$

Chính vì đây là các biến 0/1 nên các công thức trở thành các chương trình số nguyên; tất cả các ràng buộc khác đều hoàn toàn tuyến tính. Cụ thể, mục tiêu trong chương trình là giảm thiểu độ dài chuyến tham quan

$$\sum_{i=1}^{n} \sum_{j \neq i,\, j=1}^{n} c_{ij} x_{ij}.$$

Nếu không có thêm ràng buộc nào nữa, $\{x_{ij}\}_{i,j}$ sẽ thực sự bao phủ tất cả các tập hợp con của tập hợp các cạnh, rất xa so với các tập hợp các cạnh trong 1 chuyến tham quan & cho phép 1 giá trị tối thiểu tầm thường trong đó tất cả $x_{ij} = 0$. Do đó, cả hai công thức đều có ràng buộc là tại mỗi đỉnh, chỉ có đúng 1 cạnh vào & 1 cạnh ra, có thể được biểu thị dưới dạng $2n$ phương trình tuyến tính

$$\begin{cases} \sum_{i=1,\, i \neq j}^{n} x_{ij} = 1, \ \forall j \in [n], \\ \sum_{j=1,\, j \neq i}^{n} x_{ij} = 1, \ \forall i \in [n]. \end{cases}$$

Những ràng buộc này đảm bảo rằng tập hợp các cạnh được chọn trông giống như 1 chuyến tham quan cục bộ, nhưng vẫn cho phép các giải pháp vi phạm yêu cầu toàn cục là có *1* chuyến tham quan đi qua tất cả các đỉnh, vì các cạnh được chọn có thể tạo thành 1 số chuyến tham quan, mỗi chuyến tham quan chỉ đi qua 1 tập hợp con các đỉnh; có thể nói, chính yêu cầu toàn cục này khiến TSP trở thành 1 bài toán khó. Các công thức MTZ & DFJ khác nhau ở cách chúng thể hiện yêu cầu cuối cùng này dưới dạng các ràng buộc tuyến tính.

6.3.1 Miller–Tucker–Zemblin formulation

6.3.2 Dantzig–Fulkerson–Johnson formulation

6.4 Computing a solution of TSP

The traditional lines of attack for the NP-hard problems are the following:

- Devising exact algorithms, which work reasonably fast only for small problem sizes.
- Devising "suboptimal" or heuristic algorithms, i.e., algorithms that deliver approximated solutions in a reasonable time.
- Finding special cases for the problem ("subproblems") for which either better or exact heuristics are possible.

– Các hướng tấn công truyền thống cho các bài toán NP-hard như sau:

- Thiết kế các thuật toán chính xác, chỉ hoạt động khá nhanh đối với các bài toán có kích thước nhỏ.
- Thiết kế các thuật toán "không tối ưu" hoặc thuật toán heuristic, tức là các thuật toán đưa ra các giải pháp gần đúng trong thời gian hợp lý.
- Tìm các trường hợp đặc biệt cho bài toán ("các bài toán con") mà có thể áp dụng các heuristic tốt hơn hoặc chính xác hơn.

6.4.1 Exact algorithms

The most direct solution would be to try all permutations (ordered combinations) & see which one is cheapest (using brute-force search). The running time for this approach lies within polynomial factor of $O(n!)$, the factorial of the number of cities, so this solution becomes impractical even for only 20 cities.

– Giải pháp trực tiếp nhất là thử tất cả các hoán vị (các tổ hợp có thứ tự) & xem cái nào rẻ nhất (sử dụng tìm kiếm theo kiểu brute-force). Thời gian chạy cho cách tiếp cận này nằm trong hệ số đa thức của $O(n!)$, giai thừa của số thành phố, do đó giải pháp này trở nên không thực tế ngay cả khi chỉ có 20 thành phố.

1 of the earliest applications of dynamic programming is the Held–Karp algorithm, which solves the problem in time $O(n^2 2^n)$. This bound has also been reached by Exclusion–Inclusion in an attempt preceding the dynamic programming approach.

– 1 trong những ứng dụng sớm nhất của lập trình động là thuật toán Held-Karp, giải quyết vấn đề trong thời gian $O(n^2 2^n)$. Giới hạn này cũng đã đạt được bởi Exclusion-Inclusion trong 1 nỗ lực trước phương pháp quy hoạch động.

Improving these time bounds seems to be difficult. E.g., it has not been determined whether a classical exact algorithm for TSP that runs in time $O(1.9999^n)$ exists. The currently best quantum exact algorithm for TSP due to Ambainis et al. runs in time $O(1.728^n)$.

– Việc cải thiện các giới hạn thời gian này có vẻ khó khăn. Ví dụ, vẫn chưa xác định được liệu có tồn tại thuật toán chính xác cổ điển cho TSP chạy trong thời gian $O(1.9999^n)$ hay không. Thuật toán chính xác lượng tử tốt nhất hiện tại cho TSP do Ambainis và cộng sự thực hiện chạy trong thời gian $O(1.728^n)$.

Other approaches include:

- Various branch-&-bound algorithms, which can be used to process TSPs containing thousands of cities.
- Progressive improvement algorithms, which use techniques reminiscent of linear programming. This works well for up to 200 cities.

- Implementations of branch-&-bound & problem-specific cut generation (branch-&-cut); this is the method of choice for solving large instances. This approach holds the current record, solving an instance with 85900 cities.

– Các phương pháp tiếp cận khác bao gồm:

- Nhiều thuật toán nhánh-&-bound, có thể được sử dụng để xử lý TSP chứa hàng nghìn thành phố.
- Các thuật toán cải tiến tiến bộ, sử dụng các kỹ thuật gợi nhớ đến lập trình tuyến tính. Điều này hoạt động tốt với tối đa 200 thành phố.
- Các triển khai của thế hệ cắt nhánh-&-bound & cụ thể cho vấn đề (branch-&-cut); đây là phương pháp được lựa chọn để giải quyết các trường hợp lớn. Phương pháp tiếp cận này giữ kỷ lục hiện tại, giải quyết 1 trường hợp với 85900 thành phố.

An exact solution for 15112 German towns from TSPLIB was found in 2001 using the cutting-plane method proposed by George Dantzig, Ray Fulkerson, & Selmer M. Johnson in 1954, based on linear programming. The computations were performed on a network of 110 processors located at Rice University & Princeton University. The total computation time was equivalent to 22.6 years on a single 500 MHz Alpha processor. In May 2004, the TSP of visiting all 24978 towns in Sweden was solved: a tour of length approximately 72500 km was found, & it was proven that no shorter tour exists. In Mar 2005, the TSP of visiting all 33810 points in a circuit board was solved using *Concorde TSP Solver*: a tour of length 66048945 units was found, & it was proven that no shorter tour exists. The computation took approximately 15.7 CPU-years. In Apr 2006 an instance with 85900 points was solved using *Concorde TSP Solver*, talking over 136 CPU-years.

– 1 lời giải chính xác cho 15112 thị trấn Đức từ TSPLIB đã được tìm thấy vào năm 2001 bằng cách sử dụng phương pháp mặt phẳng cắt do George Dantzig, Ray Fulkerson, & Selmer M. Johnson đề xuất vào năm 1954, dựa trên lập trình tuyến tính. Các phép tính được thực hiện trên 1 mạng lưới gồm 110 bộ xử lý đặt tại Đại học Rice & Đại học Princeton. Tổng thời gian tính toán tương đương với 22,6 năm trên 1 bộ xử lý Alpha 500 MHz duy nhất. Vào tháng 5 năm 2004, TSP của việc ghé thăm tất cả 24978 thị trấn ở Thụy Điển đã được giải quyết: 1 chuyến đi có chiều dài khoảng 72500 km đã được tìm thấy, & người ta đã chứng minh rằng không tồn tại chuyến đi nào ngắn hơn. Vào tháng 3 năm 2005, TSP của việc ghé thăm tất cả 33810 điểm trong 1 bảng mạch đã được giải quyết bằng cách sử dụng *Concorde TSP Solver*: 1 chuyến đi có chiều dài 66048945 đơn vị đã được tìm thấy, & người ta đã chứng minh rằng không tồn tại chuyến đi nào ngắn hơn. Việc tính toán mất khoảng 15,7 năm CPU. Vào tháng 4 năm 2006, 1 trường hợp với 85900 điểm đã được giải quyết bằng *Concorde TSP Solver*, mất hơn 136 năm CPU.

6.4.2 Heuristic & approximation algorithms

Various heuristics & approximation algorithms, which quickly yield good solutions, have been devised. These include the multi-fragment algorithm. Modern methods can find solutions for extremely large problems (millions of cities) within a reasonable time which are, with a high probability, just 2–3% away from the optimal solution.

– Nhiều thuật toán heuristic & xấp xỉ, nhanh chóng đưa ra các giải pháp tốt, đã được đưa ra. Chúng bao gồm thuật toán đa phân mảnh. Các phương pháp hiện đại có thể tìm ra giải pháp cho các vấn đề cực lớn (hàng triệu thành phố) trong thời gian hợp lý, với xác suất cao, chỉ cách giải pháp tối ưu 2–3%.

Several categories of heuristics are recognized.

1. **Constructive heuristics.** The nearest neighbor (NN) algorithm (a greedy algorithm) lets the salesman choose the nearest unvisited city as his next move. This algorithm quickly yields an effectively short route. For n cities randomly distributed on a plane, the algorithm on average yields a path 25% longer than the shortest possible path; however, there exist many specially-arranged city distributions which make the NN algorithm give the worst route. This is true for both asymmetric & symmetric TSPs. Rosenkrantz et al. showed that the NN algorithm has the approximation factor $\Theta(\log |V|)$ for instances satisfying the triangle inequality. A variation of the NN algorithm, called *nearest fragment (NF) operator*, which connects a group (fragment) of nearest unvisited cities, can find shorter routes with successive iterations. The NF operator can also be applied on an initial solution obtained by the NN algorithm for further improvement in an elitist model, where only better solutions are accepted.

   – Thuật toán láng giềng gần nhất (NN) (thuật toán tham lam) cho phép người bán hàng chọn thành phố gần nhất chưa được ghé thăm làm nước đi tiếp theo. Thuật toán này nhanh chóng tạo ra 1 tuyến đường ngắn hiệu quả. Đối với n thành phố được phân bố ngẫu nhiên trên 1 mặt phẳng, thuật toán trung bình tạo ra 1 đường đi dài hơn 25% so với đường đi ngắn nhất có thể; tuy nhiên, có nhiều phân phối thành phố được sắp xếp đặc biệt khiến thuật toán NN đưa ra tuyến đường tệ nhất. Điều này đúng với cả TSP đối xứng & không đối xứng. Rosenkrantz và cộng sự đã chỉ ra rằng thuật toán NN có hệ số xấp xỉ $\Theta(\log |V|)$ đối với các trường hợp thỏa mãn bất đẳng thức tam giác. Một biến thể của thuật toán NN, được gọi là *toán tử đoạn gần nhất (NF)*, kết nối 1 nhóm (đoạn) thành phố gần nhất chưa được ghé thăm, có thể tìm ra các tuyến đường ngắn hơn với các lần lặp liên tiếp. Toán tử NF cũng có thể được áp dụng cho 1 giải pháp ban đầu thu được bởi thuật toán NN để cải tiến hơn nữa trong mô hình ưu tú, trong đó chỉ chấp nhận các giải pháp tốt hơn.

   The bitonic tour of a set of points is the minimum-perimeter monotone polygon that has the points as its vertices; it can be computed efficiently with dynamic programming.

   – Chuyến du hành bitonic của một tập hợp các điểm là đa giác đơn điệu có chu vi nhỏ nhất có các điểm là đỉnh của nó; nó có thể được tính toán hiệu quả bằng quy hoạch động.

Another constructive heuristic, Match Twice & Stich (MTS), performs 2 sequential matchings, where the 2nd matching is executed after deleting all the edges of the 1st matching, to yield a set of cycles. The cycles are then stitched to produce the final tour.

– 1 phương pháp tìm kiếm mang tính xây dựng khác, Match Twice & Stich (MTS), thực hiện 2 phép ghép nối tuần tự, trong đó phép ghép nối thứ 2 được thực hiện sau khi xóa tất cả các cạnh của phép ghép nối thứ nhất, để tạo ra một tập hợp các chu kỳ. Các chu kỳ sau đó được khâu lại để tạo ra chuyến tham quan cuối cùng.

2. **Algorithm of** Christofides & Serdyukov**.**
3. **Pairwise exchange.**
4. k**-opt heuristic, or Lin–Kernighan heuristics.**
5. V**-opt heuristic.**
6. **Randomized improvement.**
7. **Constricting Insertion Heuristic.** This starts with a sub-tour e.g. the convex hull & then inserts other vertices.
8. **Ant colony optimization.**

6.5 Some special cases of TSP – Vài trường hợp đặc biệt của bài toán người bán hàng du lịch

7 Miscellaneous

7.1 Contributors

1. PGS. TS. Đỗ Văn Nhơn. *Slide Bài Giảng & Bài Tập môn học Introduction to Artificial Intelligence – Nhập Môn Trí Tuệ Nhân Tạo.*
2. Vòng Lý Nàm Phúc. C++ codes.

Tài liệu

[Tiệ25] Vũ Khắc Tiệp. *Machine Learning Cơ Bản*. 2025, p. 422.