

Lecture Note: Introduction to Artificial Intelligence

Bài Giảng: Nhập Môn Trí Tuệ Nhân Tạo

Nguyễn Quân Bá Hồng*

Ngày 21 tháng 5 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

1	Some Basic Concepts – Vài Khái Niệm Cơ Bản	2
1.1	Search problems – Bài toán tìm kiếm	2
1.2	Optimization problem – Bài toán tối ưu	2
1.2.1	Search space – Không gian tìm kiếm	3
1.2.2	Continuous optimization problem – Bài toán tối ưu liên tục	3
1.2.3	Combinatorial optimization problem – Bài toán tối ưu tổ hợp	3
1.3	Search algorithms – Các thuật toán tìm kiếm	4
1.3.1	Applications of search algorithms – Ứng dụng của thuật toán tìm kiếm	4
1.3.2	Classes of search algorithms	5
1.4	Gradient – Độ dốc	6
1.5	Minimax	8
2	Heuristic Algorithms – Các Thuật Giải Heuristic	8
2.1	Heuristic (Computer Science) – Tự tìm tòi (Khoa Học Máy Tính)	8
2.1.1	Pitfalls of heuristic – Những cạm bẫy của phương pháp tìm kiếm	10
2.2	Admissible heuristics – Các phương pháp tìm kiếm có thể chấp nhận được	10
2.2.1	Optimality proof of admissible heuristics	11
2.3	Consistent heuristic	12
3	Scheduling problem – Bài toán phân công	12
3.1	Optimal job scheduling – Lên lịch công việc tối ưu	13
3.1.1	Single-stage jobs vs. multi-stage jobs – Công việc 1 giai đoạn so với công việc nhiều giai đoạn	13
3.1.2	Machine environments – Môi trường máy móc	13
3.1.3	Job characteristics – Đặc điểm công việc	14
3.1.4	Objective functions – Các hàm mục tiêu	15
3.2	Fractional job scheduling – Lập lịch công việc theo từng phần	15
3.3	Bài toán tô màu đồ thị – Graph coloring problem	16
3.4	Shortest path problem – Bài toán đường đi ngắn nhất	16
3.5	Traveling salesman problem (TSP) – Bài toán người bán hàng du lịch	16
3.6	15 Puzzle Problem	17
4	Miscellaneous	17

*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>.

4.1 Contributors	17
Tài liệu	17

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

- $f: \mathbb{R}^n \rightarrow \mathbb{R}$: **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 logarit. 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 [minimax](#) 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 & inequalities/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) = ", f1(x))
print("df(x) = ", df1(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](https://en.wikipedia.org/wiki/List_of_derivatives).

Bài toán 2. Xét hàm số $f(x, y) = 2x^3y^2 + \frac{\sqrt{x^3}}{y} + \sin(x^2y) + e^{\cos(xy^2)}$. 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

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

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.cos(x * y**2))
    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.cos(x * y**2))
    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 [search algorithms](#) at each branching step based on available information to decide which branch 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 cốt lõi 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 $d^*(v_i, v_g)$ to the goal node v_g in a **directed graph** G containing n total nodes or vertices labeled v_0, v_1, \dots, v_n , “admissible” means roughly that the heuristic underestimates the cost to the goal or formally that $h(v_i, v_g) \leq d^*(v_i, v_g), \forall (v_i, v_g)$ where $i, g \in \{0, 1, \dots, n\}$.

– 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 $d^*(v_i, v_g)$ 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, \dots, 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 $h(v_i, v_g) \leq d^*(v_i, v_g), \forall (v_i, v_g)$ trong đó $i, g \in \{0, 1, \dots, n\}$.

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.

consistent heuristic $\not\Rightarrow$ 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 , $h^*(n)$ is the optimal cost to reach a goal from n . Then $h(n)$ is *admissible* if $h(n) \leq h^*(n)$, \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_{eval} , S_{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.:

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

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

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 $n \in \mathbb{N}^*$ công việc & các việc sẽ được thực hiện bởi $m \in \mathbb{N}^*$ 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 $t[0], \dots, t[n-1]$ với $t[i]$: thời gian để 1 máy thực hiện việc i .

Output. Bảng phân công tối ưu.

Sample.

roster.inp	roster.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, \dots, w_n (w : work), gọi m máy là M_1, \dots, 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, \dots, T_m lần lượt có số phần tử là n_1, \dots, n_m , i.e., $|T_i| = n_i, \forall i = 1, \dots, m$. □

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/roster.inp.
- Output: https://github.com/NQBH/advanced_STEM_beyond/blob/main/AI/Python/roster.out.
- Python: https://github.com/NQBH/advanced_STEM_beyond/blob/main/AI/Python/roster.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 roster – Bài toán phân công mở rộng). Có $n \in \mathbb{N}^*$ công việc & $m \in \mathbb{N}^*$ 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_roster.inp	extended_roster.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 $n \in \mathbb{N}^*$ công việc sẽ được phân công cho $m \in \mathbb{N}^*$ 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.

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. 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. **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 older 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_{lj} 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_{lj} 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 $\text{fix}_j \subset [m]$, & 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 : **Throughput**. 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.
- $R|\text{pmtn}|\sum C_i$: assigning tasks to a variable number of unrelated parallel machines, allowing preemption, minimizing total completion time.

- $J3|p_{ij} = 1|C_{\max}$: a 3-machine job shop problem with unit processing times, where the goal is to minimize the maximum completion time.
- $P|size_j|C_{\max}$: 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](#).

3.3 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. ???

3.4 Shortest path problem – Bài toán đường đi ngắn nhất

Bài toán 6. Cài đặt \mathcal{E} 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 .

3.5 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 1 (Original Traveling Salesman Problem (TSP)). *Given a list of cities \mathcal{E} the distances between each pair of cities, what is the shortest possible route that visits each city exactly once \mathcal{E} returns to the origin city?*

– Cho 1 danh sách các thành phố \mathcal{E} khoảng cách giữa mỗi cặp thành phố, đâu là tuyến đường ngắn nhất có thể đi qua mỗi thành phố đúng 1 lần \mathcal{E} 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 2 (Traveling Salesman Problem (TSP)). *The traveling salesman must visit every city in this territory exactly once \mathcal{E} 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).*

3.6 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 $t \in \mathbb{N}^*$. Với t dòng tiếp theo, mỗi dòng chứa đúng 1 hoán vị $\{a_n\}_{n=1}^{15} = a_1, a_2, \dots, a_{15}$ 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	$h(n) = 36$
4 6 3 8 7 12 9 14 15 13 1 5 2 10 11	

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

4 Miscellaneous

4.1 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 LỖ NÀM PHÚC. C++ codes for scheduling problems.

Tài liệu

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