Utku Kepir
s328174

FINAL REPORT

1. Project Work

i. Link

https://github.com/UtkuKepir/CI2024_project-work

ii. Methodology

I used only numpy library. Firstly, I defined a Node class. In this class, I implemented my tree structure to represent the data. I initialized my Node class, with value, left and right. left and right represent values(x[0], x[1]…., and 1,2,3 …) and value represents operator like sin. Left is required for single-value operators like sin(left). On the other hand, right and left are required for double value operators like '+' i.e. left + right. I have evaluate() function for the representation of the mathematical formulas. I used '+', '-', '*', '/', '^', 'sin', 'cos', 'tan', 'sigmoid', 'exp', 'log', 'sqrt', 'inv' for mathematical calculations. I tried to handle inf, Nan, and very large values here properly. For example, in the below function, 1000 is used for not dealing with large fitness values for my algorithm. It is an optimization for my algorithm.

elif self.value == 'exp':

    return np.exp(left) *if* np.all(left < 1000) *else* float('inf')

I used count_nodes() function for applying a penalty according to Pareto-optimality to the fitness function as a complexity. __str__ function is just for to see the tree as an string.

My main algorithm is Genetic Programming which is an Evolutionary Algorithm variant. In addition to the Genetic Programming, I used Simulated Annealing for better exploration. I used 0.9 as a cooling rate for Simulated Annealing to train faster. Higher cooling rate means slower training. I prefer to continue with Modern Genetic Programming. I used tree structure for representation, recombination for exchange of elements, steady-state model as a population model, roulette wheel(fitness proportional) selection for parent selection, and function with the best fitness value(the lowest MSE) is used for deterministic survivor selection. Firstly, I used mutation with low probabilities, but I realized that it did not bring me any better result, so in the end I decided not to use mutation. Roulette wheel parent selection is used for decreasing selective pressure. Also, I added a penalty as a complexity(number of used node) to the MSE value while calculating fitness. For evolving, I am initializing my population with random tree. Firstly, I am taking care of using all rows of the x_train. Because I need x[0], x[1],.. at least once in my funciton as you can see below.

```
operator_choices = np.array(['+', '-', '*', '/', 'sin', 'cos', 'tan', 'sigmoid', 'exp', 'inv'])
operator = np.random.choice(operator_choices)
if operator in ['+', '-', '*', '/']:
    left = self.generate_random_tree(depth - 1, shape, required_vars)
    right = self.generate_random_tree(depth - 1, shape, required_vars)

    if required_vars:
        missing_var = required_vars.pop()
        right = Node('+', left=right, right=Node(missing_var))

    return Node(operator, left=left, right=right)
```

After generating these variables once, I'm generating these variables again or random constants as you can see below

```
if depth == 0:
    if required_vars:
        choice = required_vars.pop()
        return Node(choice)
    else:
        if np.random.random() < 0.05:
            choice = 'x[' + str(np.random.randint(0, shape)) + ']'
            return Node(choice)
        else:
            choice = np.random.random_sample()
            return Node(float(choice))
```

Then, I am starting evolving generations *for* generation *in* range(*self*.num_gen). For each generation, I'm taking half of the population used for parent selection. Then, picking 20 parents over the population, then sorting them according to their fitness. Then, using roulette wheel, for decreasing selective pressure, I am picking 2 parents. Then parents go crossover to construct offspring. After that offspring's fitness values compete with the parents' fitness values to select a survivor. This is a steady-state population model actually. If the offspring's fitness value is lower than the population's best fitness value offspring is the survivor(best_individual). I am saving the best individual coming from parent or offspring to the population instead of the oldest individual for not losing the best individual with the best fitness value. Also, I'm adding 50 new individuals for each generation to increase diversity, and this method gave me better results than mutation. Finally, I wrote best_individuals with the best fitness values(lowest MSE) to the s328174.py.

iii. Results

Data	Fitness	Expression	Expression using Numpy	MSE using symreg.ipynb
problem_1.npz	0.1	sin(x[0])	np.sin(x[0])	7.12594e-32
problem_2.npz	29575308429996.04	inv(sigmoid((x[0] + (x[1] + x[2]))))	1 + np.exp(-(x[0] + x[1] + x[2]))	2.9617e+15
problem_3.npz	1323.05	cos(tan(x[0])) + (inv(sigmoid(x[1])) + x[2])	np.cos(np.tan(x[0])) + (1 + np.exp(-x[1])) + x[2]	132285
problem_4.npz	21.58	exp(cos((x[0] * x[1])))	np.exp(np.cos(x[0] * x[1]))	2138.15
problem_5.npz	0.15	(cos(x[0]) / (x[1] / 0.02871653301747079))	np.cos(x[0]) * 0.02871653301747079 / x[1]	30.98
problem_6.npz	5.86	cos(exp(sigmoid(x[0]))) + x[1]	np.cos(np.exp(1 / (1 + np.exp(-x[0])))) + x[1]	561.83
problem_7.npz	626.07	exp(x[0] * x[1])	np.exp(x[0] * x[1])	62592.7
problem_8.npz	21321683.32	((exp(x[5]) * (exp(x[0]) + x[3])) - (tan(inv(x[1])) + x[2])) + x[4]	((np.exp(x[5]) * (np.exp(x[0]) + x[3])) - (np.tan(1 / x[1]) + x[2])) + x[4]	2.13217e+09

iv. Possible Improvements in the Future

Diversity can be increased for reaching better fitness values. For doing this, extinction method might be used upon convergence. Also, lexicase selection may be used instead of roulette wheel selection, maybe it improves the solution. Additionally, other fine-tuning methods such as constant optimizations might be added necessary parts of the code properly.

2. Lab0

Link: https://github.com/UtkuKepir/CI2024_lab0

This Lab was just for creating GitHub repos and learning how to create an issue for another person's repos.

3. Lab1

Link: https://github.com/UtkuKepir/CI2024_lab1

The solution space depends on how you defined fitness and represent the data. I implemented 3 different fitness function for this lab. I firstly tried single mutation. Then, I realized that multiple mutations for each call performs better for this problem after trying 2 different multiple mutation functions. Thus, I continued with multiple mutations. Then, I started this lab evolution with pure greedy algorithm. Then, I added simulated annealing for the Evolutionary Algorithm to reach better exploration. I've adjust the hyperparameters such as temperature, maximum number of improvements, Num_Sets and Universe_Size to reach smooth landscape near the global optimum(lower fitness). You can find my results below.

CI2024_lab1

Instance	Universe_Size	Num_Sets	Density	Fitness_First	Result=Fitness_Last
1	100	10	0.2	-149.22	-24.03
2	1000	100	0.2	-34483.51	-5332.30
3	10000	1000	0.2	-4275110.48	-128273.64
4	100000	10000	0.1	-251258613.78	-95337418.41
5	100000	10000	0.2	-538410101.56	-202046181.05
6	100000	10000	0.3	-841100093.06	-316753946.60

Utku Kepir
s328174

4. Lab1 – Reviews I Received

i. Good solution, the multiple mutation performs well on this problem, and if you want i think you can try to do some small adjustments to make it perform even better. For example, you can try different starting solutions (instead of starting with all the sets selected) or fine tuning the multiple mutation function (i'm talking about the parameter 0.01) depending on the instance. In my case i also adopted a way to count how many items are covered (this helps if you select fewer sets as a starting solution) and i found that to be quite beneficial to the solution

ii. Good implementation! However, there are a few aspects that can improve the solution. It would be better to have a single program that iterates over the possible instances instead of rewriting all the code for every instance. This would improve code maintainability and readability. In the multiple tweaking process, the changing rate of 0.01 is somewhat of a "magic number." Instead, it can be treated as a hyperparameter that can be tuned to achieve better results. A larger changing rate at the beginning could favor exploration, while a smaller rate towards the end would promote exploitation. The initial solution could start with fewer sets included instead of including all sets. However, this would require some adjustments to the validity check to ensure that all elements in the universe are still covered.

5. Lab2

Link: https://github.com/UtkuKepir/CI2024_lab2

I started this lab with pure greedy algorithm. Then, I added simulated annealing for the Evolutionary Algorithm to reach better exploration. I used inversion mutation as a mutation and inver-over crossover for the crossover. I've adjust the hyperparameters such as temperature, maximum number of improvements, and max-steps to reach smooth landscape near the global optimum(lower fitness). You can find my results below for different countries.

```python
if delta_fitness > 0 or np.random.random() < np.exp(delta_fitness / temperature):
    solution = new_solution
    solution_fitness = new_fitness
    no_improvement_steps = 0

    # Update the best solution found
    if solution_fitness[1] > best_fitness[1]:
        best_solution = solution
        best_fitness = solution_fitness
        best_step = step
        best_cost = -float(solution_fitness[1])

else:
    no_improvement_steps += 1

if no_improvement_steps > max_no_improvement:
    print("Restarting from a new random solution...")
    solution = greedy_tsp_initialization(CITIES, DIST_MATRIX)
    solution_fitness = fitness(solution, num_cities)
    no_improvement_steps = 0

# Reduce the temperature
if delta_fitness > 0:
    temperature *= 0.965
else:
    temperature *= 0.98

history.append(float(solution_fitness[1]))
```

CI2024_lab2

CI2024_lab2

I've used inversion mutation, inver-over crossover and simulated-annealing algorithm for the evaluationary algorithm. I've adjust the hyperparameters such as temperature, maximum number of improvements, and max-steps to reach smooth landscape near the global optimum(lower fitness). You can find my results below for different countries.

Country	# of Cities	Initial fitness	Best Fitness	Best Step
China	726	-63962.91	-56173.65	997284
Italy	46	-4436.03	-4181.61	612937
Russia	167	-42334.16	-36672.64	84105
US	326	-48050.02	-41965.19	511396
Vanuatu	8	-1475.52	-1345.54	509

6. Lab2 – Reviews I Received

Here we have initially, a greedy algorithm that generates a simple solution by iterating over cities and selecting the nearest unvisited one, though this method is known for providing suboptimal results. The fitness of each solution is evaluated using a distance matrix that calculates the total travel distance based on geodesic distances between cities. The inversion mutation modifies the route by reversing a randomly selected segment, introducing diversity and ensuring valid routes. The inver-over crossover combines features from two parent routes, using inversion to maintain diversity while carefully ensuring that no cities are duplicated. The simulated annealing algorithm refines the initial solution by iterating over a set number of steps, accepting better solutions or those with a certain probability based on temperature, which decreases over time. This allows the algorithm to escape local optima. The approach also incorporates random restarts if no improvement is observed over a large number of steps, ensuring continuous exploration of the

solution space. Results show that the algorithm significantly improves fitness compared to the initial solution, particularly for larger datasets, though the computational complexity increases with the number of cities. While the method is robust, computationally expensive, and relies on manual parameter tuning, it offers a flexible and effective approach for solving TSP. The combination of greedy initialization and simulated annealing, along with the various mutation and crossover strategies, provides a thorough exploration of potential solutions, although there is room for improvement in terms of efficiency and automated parameter optimization.

7. Lab3

Link: https://github.com/UtkuKepir/CI2024_lab3

I've used A* Search with Linear Conflict(extension of Manhattan Distance), A* Search with Manhattan Distance, A* Search with Hamming Distance, Breadth-First Search, and Depth-First Search with Bound for the puzzle problem. You can find my results below corresponding to different methods. A* Search with Linear Conflict(Improved Manhattan Distance) gives the most efficient results. When the PUZZLE_DIM is high, Breadth-First Search and Depth-First Search with Bound are not efficient. Note that Quality and the Cost depend on Initial State. In Breadth-first is an uninformed search method the root node is expanded, then its successors, then their successors, and so on. It is a systematic approach and always complete. Breadth-first search method is optimal when all costs are equal; thus, for this lab, it is not the optimal choice. Depth-first search with Bound is an uninformed method and it is not the optimal choice for this lab. Also, it is not complete unless iterative deepening is also used. A* Search is an informed method, and it always computes the path with minimum cost by expanding a minimum number of nodes. Thus, it is an optimal choice for this lab as you can see from the results.

Utku Kepir
s328174

I've used A* Search with Linear Conflict(extension of Manhattan Distance), A* Search with Manhattan Distance, A* Search with Hamming Distance, Breadth-First Search, and Depth-First Search with Bound for the puzzle problem. You can find my results below corresponding to different methods. A* Search with Linear Conflict(Improved Manhattan Distance) gives the most efficient results. When the PUZZLE_DIM is high, Breadth-First Search and Depth-First Search with Bound are not efficient. Note that Quality and the Cost depend on Initial State.

Strategy	RANDOMIZE_STEPS	PUZZLE_DIM fitness	Quality(# of actions in the solution)	Cost(Total number of actions evaluated)
A* Search with Linear Conflict(Informed Strategy)	100_000	4	46	180595
A* Search with Manhattan Distance(Informed Strategy)	100_000	4	46	1414443
A* Search with Hamming Distance(Informed Strategy)	100_000	4	46	>>1414443 - Not efficient(2 hour passed then I stopped the execution)
Breadth-First Search(Uninformed Strategy)	100_000	4	46	>>1414443 - Not efficient(2 hour passed then I stopped the execution)
Depth-First Search with Bound(Uninformed Strategy)	100_000	4	46	>>1414443 - Not efficient(2 hour passed then I stopped the execution)

A* Search with Linear Conflict(Informed Strategy)	100_000	3	24	572
A* Search with Manhattan Distance(Informed Strategy)	100_000	3	24	1087
A* Search with Hamming Distance(Informed Strategy)	100_000	3	24	28595
Breadth-First Search(Uninformed Strategy)	100_000	3	24	130601
Depth-First Search with Bound(Uninformed Strategy)	100_000	3	24	28595 - When I decreased bound from 60 to 24, I obtained a very time efficient result
A* Search with Linear Conflict(Informed Strategy)	100_000	2	2	3
A* Search with Manhattan Distance(Informed Strategy)	100_000	2	2	3
A* Search with Hamming Distance(Informed Strategy)	100_000	2	2	3
Breadth-First Search(Uninformed Strategy)	100_000	2	2	5
Depth-First Search with Bound(Uninformed Strategy)	100_000	2	2	5

8. Lab3 – Reviews I Received

i. Overview

The proposed code presents a very good solution with a lot of different approaches such as A*, DFS and BFS, each one with different strategies as well.

Code and documentation

The code is well organized, the comment are not a lot but I think they are sufficient to understand the goal of each section. The code ran successfully with no bugs and the results output is very clear, well done.

The readme file is simple and well structured, thank you for compiling all the results in the table, it helps a lot to make comparisons between the different strategies.

Possible improvements

- On the side of the algorithms I have nothing to suggest, I think you explored the strategies to solve the problem in a more than excellent way, undoubtedly more than I did.
- I suggest to use a "requirements.txt" file to install all the dependencies at once using "pip install -r requirements.txt".
- In the readme file there are three image links that don't work, maybe you forgot to upload the png files in the repo.

Final comment

Very good job! Kudos 👍

ii. I like your approach of trying multiple ways to solve this n-puzzle problem.
You have a good solution for the problem.
Results are presented in a clear manner in the readme.
A small complaint, it 's that i would suggest you to comment more your code, some part are difficult to understand .
An other small complaint is that your algorithm doesn't deal with the edge case where the starting state is identical to the goal state, but it's only a problem of how you structured the if statement to print .
This is the
culprit final_state_a_star_linear_conflict,path_a_star_linear_conflict,quality_a_star_linear_conflict,cost_a_star_linear_conflict = a_star(state, h_linear_conflict)
if path_a_star_linear_conflict:
print(f"Steps (quality): {quality_a_star_linear_conflict}, Total states evaluated (cost){cost_a_star_linear_conflict},Finalstate:
{final_state_a_star_linear_conflict}, A* solution:
{path_a_star_linear_conflict}")
else:
print("A* linear conflict solution not found")`
You get an empty path_a_star_linear_conflict in this case and so your algorithm tells "A* linear conflict solution not found"
Create sub-issue

9. Lab3F

Link: https://github.com/UtkuKepir/CI2024_lab3F

I used nx.shortest_path function to reach the solution. This function uses Dijktra's Algorithm to find shortest path.

Lab3

In [61]:
```
median = np.median(DIST_MATRIX.reshape(1, -1))
ic(median)
# DIST_MATRIX[DIST_MATRIX > median] = np.inf
G = nx.Graph()
for c1, c2 in combinations(CITIES.itertuples(), 2):
    G.add_node(c1.Index, name=c1.name)
    G.add_node(c2.Index, name=c2.name)
    if DIST_MATRIX[c1.Index, c2.Index] <= median: #simple tweaking: if the distance btw two cities > median, then make the
        G.add_edge(c1.Index, c2.Index, weight=DIST_MATRIX[c1.Index, c2.Index])
nx.is_connected(G)
```

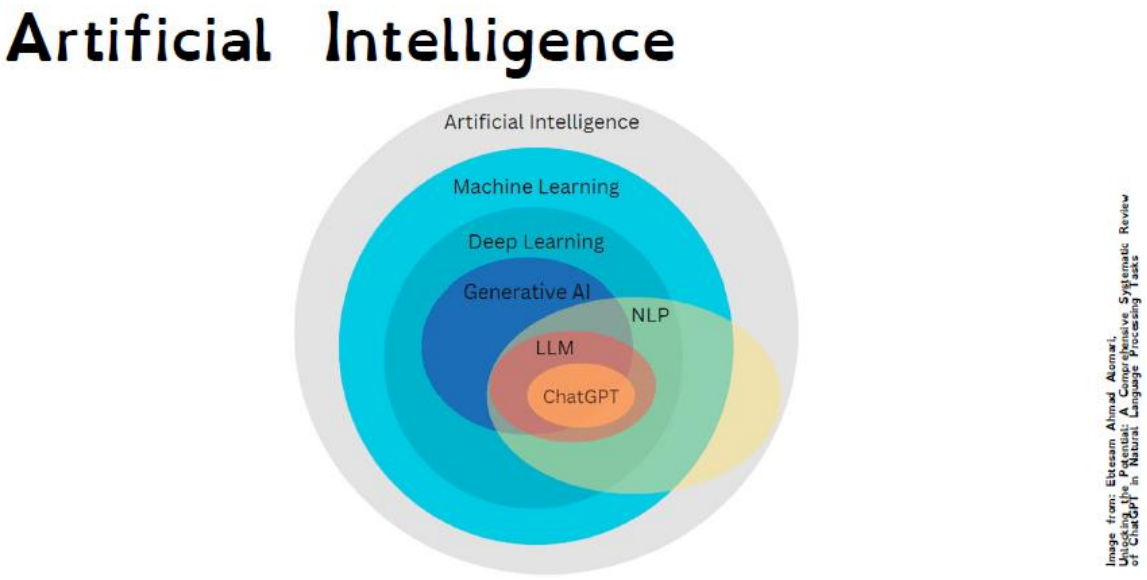
```
ic| median: np.float64(367.0694334013403)
```

Out[61]: True

In [62]:
```
ic(list(G.nodes(data=True))[:10])
ic(list(G.edges(data=True))[:10])

start_city = CITIES[CITIES['name'] == 'Turin'].index[0]
end_city = CITIES[CITIES['name'] == 'Bari'].index[0]

if start_city in G and end_city in G:

    shortest_path = nx.shortest_path(G, source=start_city, target=end_city, weight='weight')
    path_distance = nx.shortest_path_length(G, source=start_city, target=end_city, weight='weight')
    ic("Shortest path:", [CITIES.loc[node, 'name'] for node in shortest_path])
    ic("Path distance:", path_distance)
else:
    print("Selected cities are not in the connected Graph")
```

```
ic| list(G.nodes(data=True))[:10]: [(0, {'name': 'Ancona'}),
                                    (1, {'name': 'Andria'}),
                                    (2, {'name': 'Bari'}),
                                    (3, {'name': 'Bergamo'}),
                                    (4, {'name': 'Bologna'}),
                                    (5, {'name': 'Bolzano'}),
                                    (6, {'name': 'Brescia'}),
                                    (7, {'name': 'Cagliari'}),
                                    (8, {'name': 'Catania'}),
                                    (9, {'name': 'Ferrara'})]
ic| list(G.edges(data=True))[:10]: [(0, 1, {'weight': np.float64(349.30401144305745)}),
```

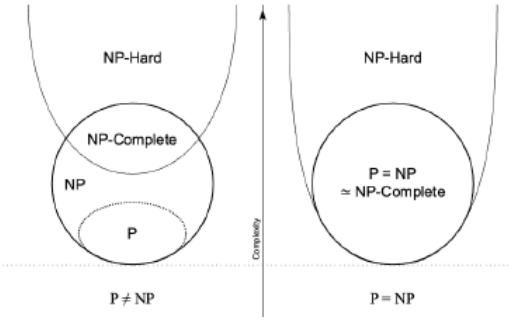
10. Notes for the Project and Exam



Utku Kepir
s328174

NP Problems

- Nondeterministic Polynomial time problems
  - Verifiable in polynomial time
  - Infamous P vs. NP, NP vs. co-NP dilemmas
- NP-Hard, NP-Complete
- Space/Time complexity
- Practical definition:
  - You can only test a small fraction of the input space



Local Search Algorithms: Techniques used to find a better solution by iteratively improving an existing solution. Useful for solving complex optimization problems where finding an exact solution is computationally expensive. Paradigmatic example: Gradient Descent, Stochastic Gradient Descent, Hill Climbing, Simulated Annealing, Tabu Search, Genetic Algorithms, Particle Swarm Optimization

Fitness: The quantitative representation of natural and sexual selection within evolutionary biology.

Fitness = - Cost

Maximize the fitness or minimize the cost.

Global and Local Optima: The highest peak in the landscape is the global optimum, representing the highest fitness. Local optima are peaks that are not as high as the global optimum, but still represent relatively high fitness (not really formal, indeed)
.Solutions are attracted by optima (basins of attraction in dynamical systems).

Valleys: Valleys are regions of low fitness, they can be deep or shallow, and they can be separated by plateaus or ridges.

Utku Kepir
s328174

Plateaus: Vast areas of uniformly high fitness.
Ridges: Ridges are elevated areas that connect different peaks.
Funnel: Region where the fitness increases rapidly towards a particular point.

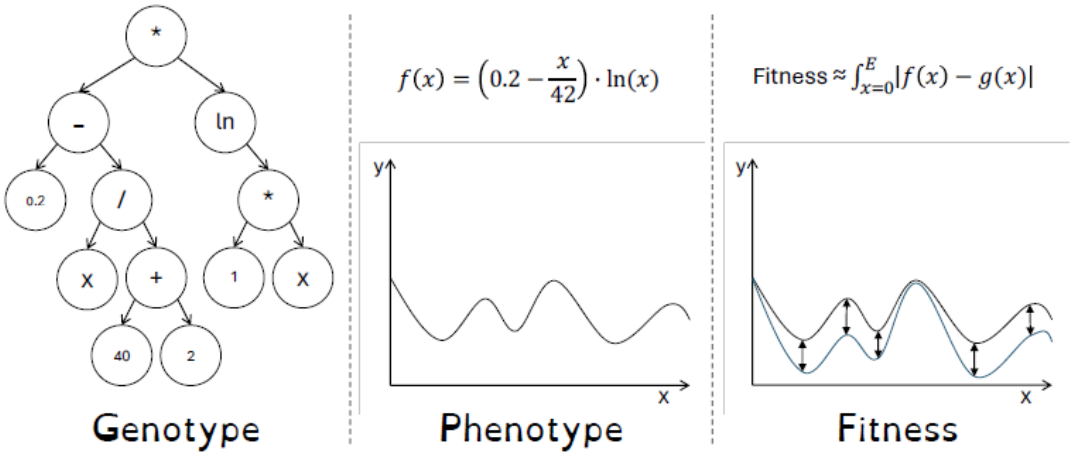
Known Problems: Deception: A fitness function that "intentionally" misleads the optimization process. Local optima are more attractive than the global optimum. **In general, we increase the step size to take from the local maximums, in cases where we do not do this, we can use simulated annealing, tabu search or iterated local search to find the global optimum. The simulated annealing algorithm works best if the temperature is reduced at a slow rate, so this value should be close to 1. (but values closer to 1 will also cause the algorithm to run longer).**

Exploration: Seeking out new information. Delving into the unknown to discover something new.
Exploitation: Choosing the best-known option based on existing information to maximize immediate benefits.

Fitness: how well the candidate solution is able to solve the target problem
Genotype: the internal representation of the individual, i.e., what is directly manipulated by genetic operators
Phenotype: the candidate solution that is encoded in the genotype

Utku Kepir
s328174

Levels (symbolic regression)



Modeling is going "from the problem space to the fitness landscape" = Genotypes are "mapped" to phenotypes

Tweaking is just changing one random bit

Greedy algorithms are a class of algorithms that make locally optimal choices at each step with the hope of finding a global optimum solution. In these algorithms, decisions are made based on the information available at the current moment without considering the consequences of these decisions in the future. The key idea is to select the best possible choice at each step, leading to a solution that may not always be optimal but is often good enough for many problems.

Recombination is an exploration on the other hand mutation is exploitation. Recombination mixes together two or more solutions to create the offspring on the other hand mutation performs(usually small) change in an individual.

Utku Kepir
s328174

Premature convergence

- I.e., the tendency of an algorithm to converge towards a point where it was not supposed to converge to in the first place
- Probably an oxymoron
- Holland's "Lack of speciation"
- EAs general inability to exploit environmental niches

Niches

- Niches favor the divergence of character
- Niches and speciation
- How to create "niches" in EAs since the environment is missing?

To handle premature convergence you can increase the population for each step to increase the diversity.

How diversity is promoted (practice)

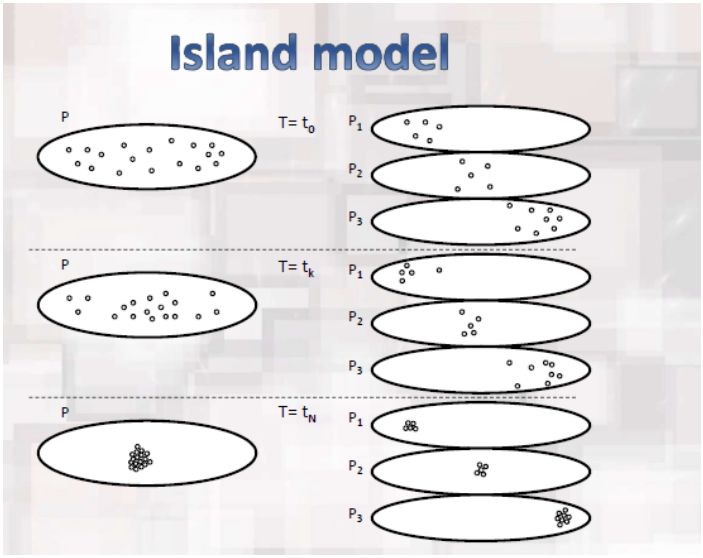
- Fitness scaling
- Fitness holes
- Tweaking selection mechanism
- Adding selection mechanism
- Multiple populations
- Population topologies

Utku Kepir
s328174

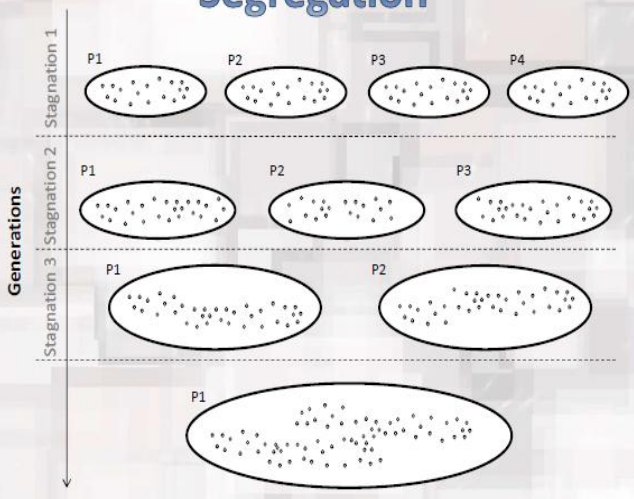
Type of selection

- **Parent** selection (α)
  - Usually non-determinstic
- **Survival** selection (ω)
  - Usually deterministic

Island model



Segregation



Random immigrants

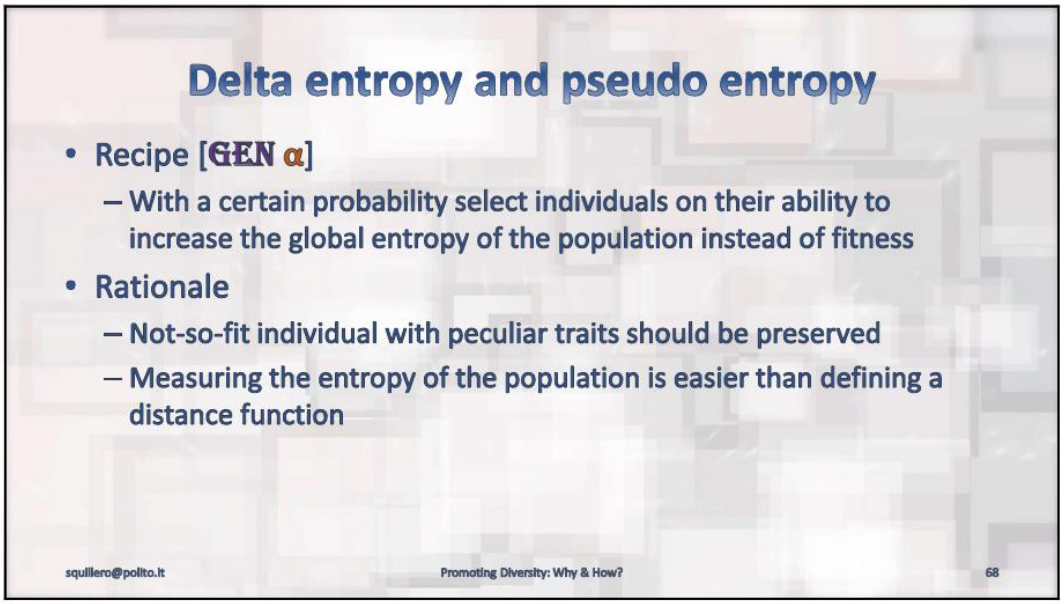
- Recipe [PHE αω]
  - Periodically insert random individuals in the population
- Rationale
  - Try to introduce novelty
- Caveats
  - Newborns may need to be artificially kept alive when competing against already optimized individuals

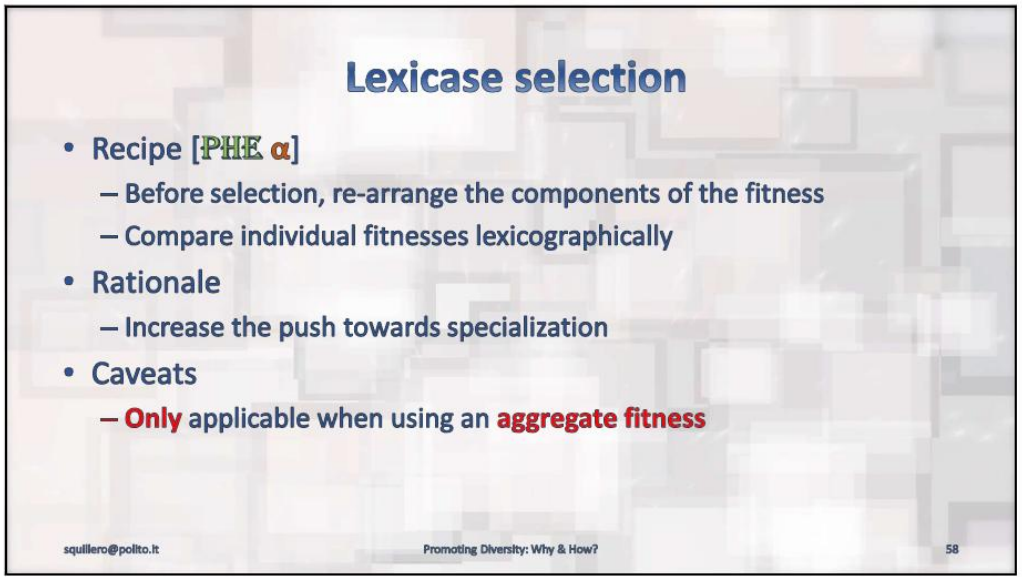
Diversifiers

- Recipe [GEN αω]
  - Detect less populated areas in the search space and try to generate random inhabitants
- Rationale
  - Increase variability in the gene pool regardless the fitness
  - Require a reliable distance metric

Extinction

- Recipe [PHE ω]
  - Upon convergence (or periodically) remove a significant part of the population
  - Then fill up the population with the offspring of the survivors and/or random individuals
- Rationale
  - A gust of fresh air: already optimized individuals are not enough to occupy the whole population and newborns may start exploring new regions
- Caveat
  - Fitness variability used as phenotype variability



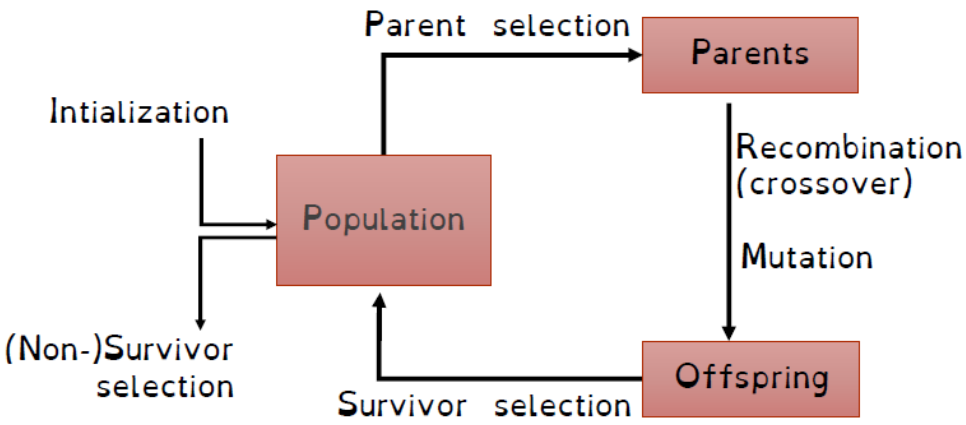


The state can be infinite, but the node has to be finite and remembered. When you put it into your state algorithm, it becomes a node!

Inversion of generational model = steady state model

Utku Kepir
s328174

EA Components



Novelty vs. Quality

- Genetic operators
- Increases population diversity
- Push towards **novelty**

- Selection
  - of parents
  - of survivors
- Decreases population diversity
- Push towards **quality**

Selection Pressure

- Qualitative vs. Quantitative measure
- Takeover time τ^*
  - Number of generations until the application of selection completely fills the population with copies of the best individual

Classic Genetic Programming

- Key elements
  - Representation: tree structures
  - Recombination: exchange of subtrees
  - Mutation: random change in trees
  - Population model: generational
  - Parent selection: fitness proportional
  - Survivor selection: deterministic

Historical EA variants

- Genetic Algorithms
- Evolution Strategies
- Evolutionary Programming
- Genetic Programming
- Learning Classifier Systems

The mutation rate in Symbolic Regression

Mutation vs. Recombination

- Performed in alternative: p_m, p_c
- Low mutation probability!
- $P_m = 0$ (Koza'92)
- $P_m = 0.05$ (Banzhaf'92)

Fitness

- Classical GP
  - % success
  - Perfectly suited for fitness proportional roulette wheel
- Symbolic Regression
  - Absolute difference from target
  - Mean Squared Error (non-linear may be better)
  - ...

Modern Genetic Programming

- Key elements
  - Representation: tree or graph structures
  - Recombination: exchange of elements
  - Mutation: random changes
  - Population model: any
  - Parent selection: any
  - Survivor selection: deterministic

Selection: Uniform

- Parents are selected by uniform random distribution whenever an operator need

$$p_i = \frac{1}{\mu}$$

- Uniform parent selection is unbiased: every individual has the same probability to be selected

Utku Kepir
s328174

Roulette wheel

- Chose an item randomly from a set according to the relative weights

$$w_0, w_1, \ldots, w_\mu$$

- Metaphor: spinning the ball in a roulette wheel with pockets of different sizes
- That is

$$p_i = \frac{w_i}{\sum_j w_j}$$

Selection: Fitness-proportional

- Probability for individual i to be selected for mating in a population size μ

$$p_i = \frac{f_i}{\sum_j f_j}$$

- I.e., roulette wheel with $w_i = f_i$
- Problems:
  - Small selection pressure in large population
  - Susceptible to function transposition

Selection: Rank-based

- Generic formula

$$w_i = r_i^p$$

- Smoothly shape selection pressure from *uniform* $(p = 0)$ to *linearized fitness rank* $(p = 1)$ or much more $(p > 1)$

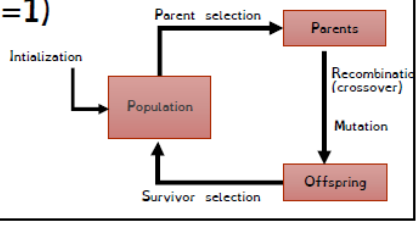
Utku Kepir
s328174

Selection: Tournament selection

- Pick τ members at random then select the best out of these
- Simple implementation, does not require sorting
- Selective pressure directly related to τ
  - $\tau = 2$ same selective pressure as linear fitness ranking
- Does not require global ordering

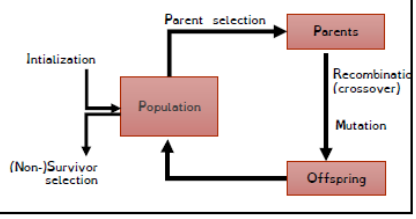
Population Management Models

- Generational model
  - Offspring replace parent, then they are selected for survival
  - ES slang: (μ,λ)
  - Typical values: $\lambda = 7\,\mu$, more recently $\lambda \approx 3\,\mu$
  - Elitist strategies: the n fittest individuals are copied unmodified as offspring (usually n=1)



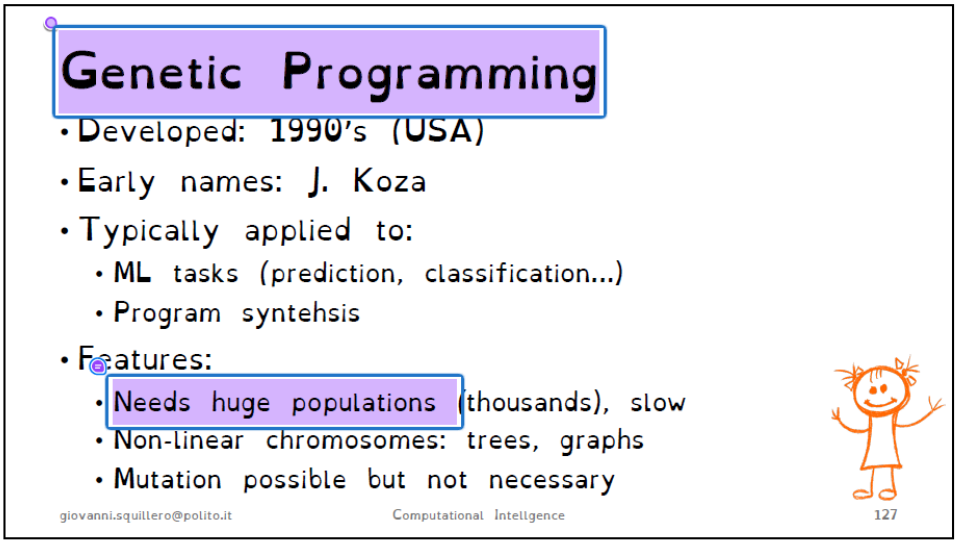
Population Management Models

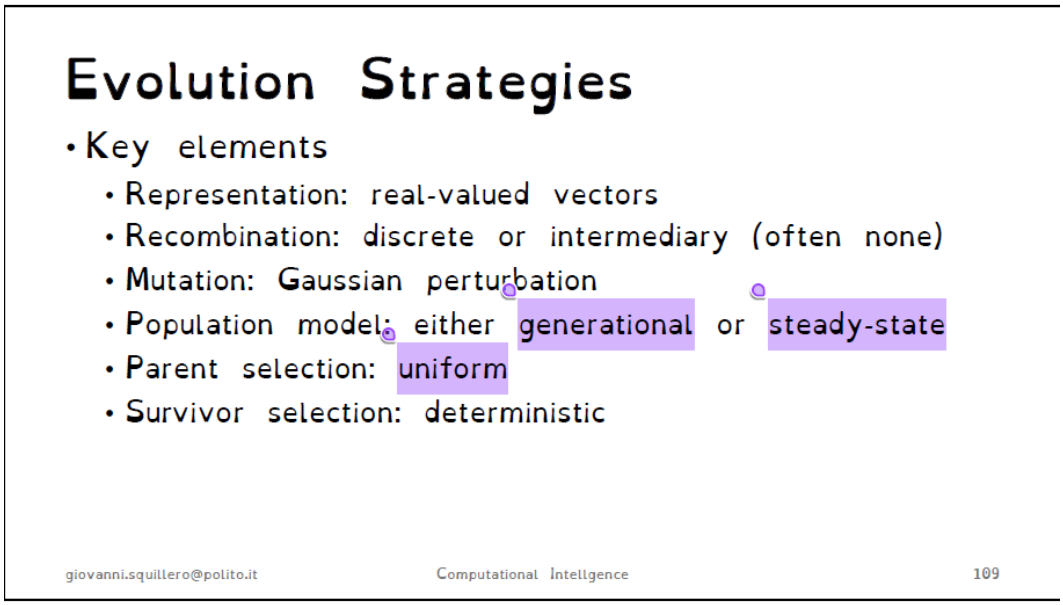
- Steady-state model
  - Offspring are added to the population, then they compete against parents for survival
  - ES slang: $(\mu+\lambda)$
  - Typical values: $\mu > \lambda$, like $\mu = 30,\ \lambda = 20$
  - Note: a $(\mu+1)$-ES is called by some scholar a "steady-state ES"



Utku Kepir
s328174

Mutation-only-EA is possible
Crossover- only-EA would not work
However, Crossover- only genetic programming work
We are searching smooth landscapes for each of the algorithms. if landscape is not smooth, you mean that the algorithms are not correct. However, while reaching a smooth lancape, variable can also be discrete or continuous.
For parameter optimization, EA, DE(Differential Evolution), PSO(Perturbation vector) are good.





ES parent selection

- Parents are selected by uniform random distribution
- ES parent selection is **unbiased**!
- Each individual has the same probability to be selected
- What about the selective pressure?

ES cave

utkuk

ONEMLI!!!!!!!! sinavda lamda buyurse selective pressure artar hoca ozellikle sinavda bunu sormayi

- $\mu > 1$ to tes
- Selective pressure indirectly tweaked by λ and μ
- Goldberg and Deb showed that takeover time

$$\tau^* = \frac{\ln \lambda}{\ln \lambda/\mu}$$

- Note: in (μ,λ)-ES
  - $\lambda > \mu$ required offspring surplus (usually $\lambda \gg \mu$)

Higher lamda means -> huge selective pressure

Evolutionary Programming

- Key elements
  - Representation: real-valued vectors
  - Recombination: none
  - Mutation: Gaussian perturbation
  - Population model: steady-state $(\mu+\mu)$
  - Parent selection: deterministic
  - Survivor selection: probabilistic q-tournament

Utku Kepir
s328174

Genetic Algorithms

- Key elements
  - Representation: fixed-length bit strings
  - Recombination: 1-point crossover
  - Mutation: multiple bit flips
  - Population model: generational
  - Parent selection: fitness proportional
  - Survivor selection: deterministic
- Note: Holland's original GA is now known as the simple genetic algorithm (SGA)

Informed strategies

- A* search
  - Best-first using function:
- f(n) = g(n) + h(n)
  - g is the actual cost
  - h is the estimated cost (heuristic)
- Complete and Optimally efficient
  - It always computes the path with minimum cost by expanding a minimum number of nodes
  - NB: Under quite reasonable assumptions

Informed strategies

- Greedy best-first
  - Expand the node with the maximum expected value
  - Not optimal
  - Incomplete (unless backtracking)

```
def h(state):
    return …

priority_function=lambda s: h(s)
```

Utku Kepir
s328174

Uninformed strategies

- Bi-directional search
  - Search forward from initial state; backward, from goal

Uninformed strategies

- Beam search
  - Uniform-cost search + limited number of nodes in each level
    (i.e., children are added to the frontier, then a certain number of nodes are trimmed from the frontier)
  - Not complete

Uninformed strategies

- Depth-limited search
  - Depth-first + limited diameter
    (i.e., children are added to the frontier, but deeper nodes are discarded)
  - Not complete unless iterative deepening is also used

Uninformed strategies

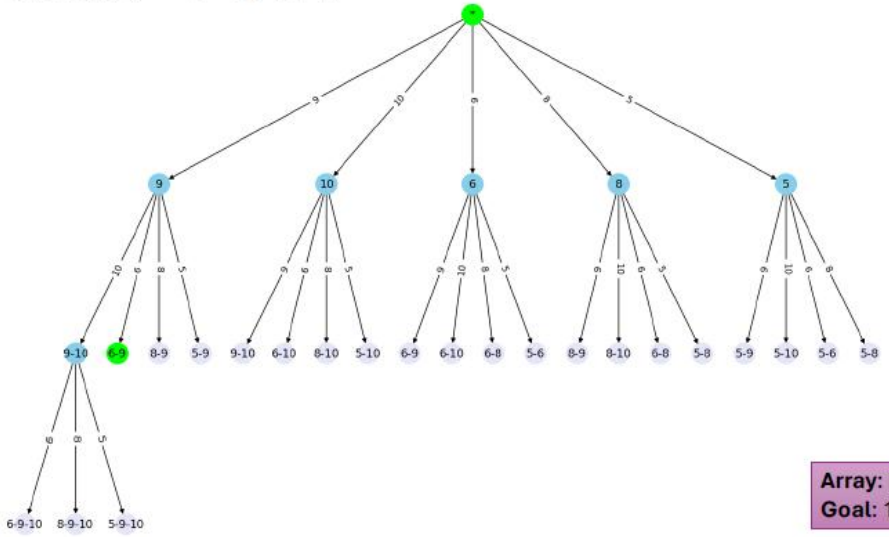
- Uniform-cost search
  - Like Breadth-first, but the node with the lowest path cost from the root is expanded
    (i.e., the frontier is a priority queue)
  - Also called Dijkstra's algorithm
  - Breadth-first can be seen as Dijkstra with unit cost

Utku Kepir
s328174

Uninformed strategies

- Breadth-first search
  - The root node is expanded, then its successors, then their successors, and so on...
    (i.e., children are added to the frontier with append)
  - Systematic
  - Always complete (if branching factor < ∞)
  - Optimal when all costs are equal

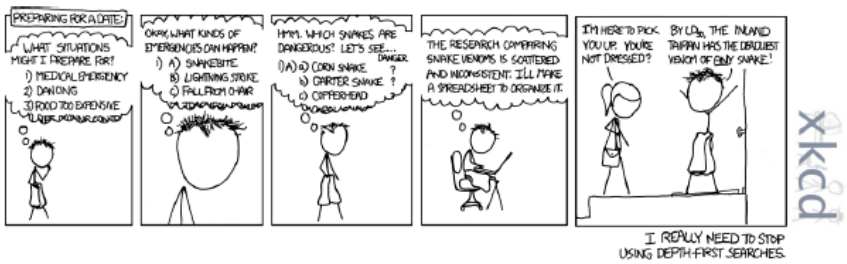
Breadth First



Array: [9, 10, 6, 8, 5]
Goal: 15

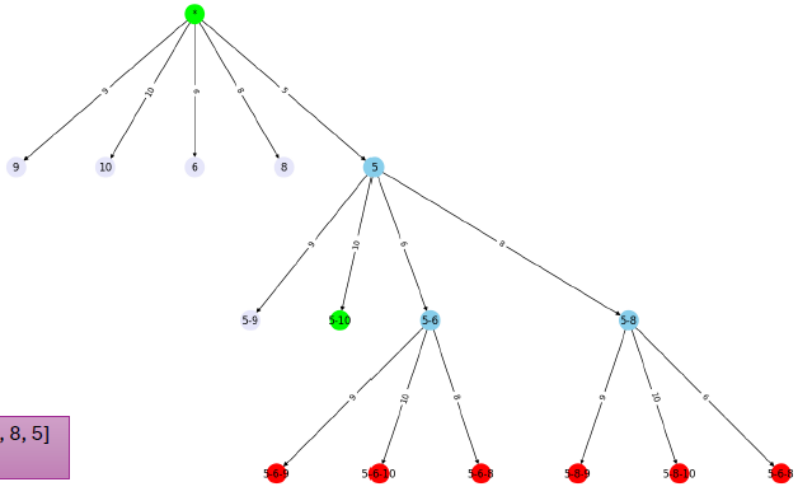
Uninformed strategies

- Depth-first search
  - The deepest node in the frontier is expanded
    Problems of cycles

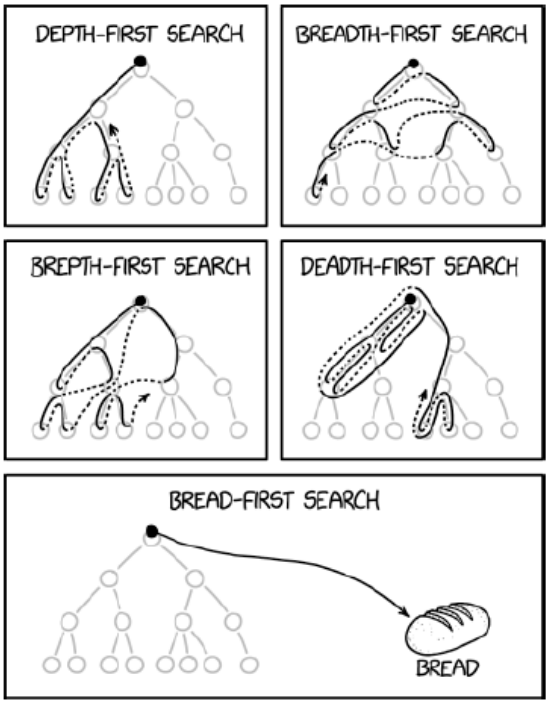


Utku Kepir
s328174

Depth First + Bound



Array: [9, 10, 6, 8, 5]
Goal: 15



Pareto-optimality

The purpose is to find a solution that is both simple and accurate.

As we have seen for lab 1 we use a Pareto frontier of both:

- accuracy MSE;
- complexity (size of the formula)