Master's degree thesis

LOG950 Logistics

RadPathFinder: An application for finding optimal paths in a radiation enviroment

Denis Senokosov

Rebecca Nerland

Number of pages including this page: 113

Molde, 23. May 2022

Molde University College
Specialized University in Logistics

Mandatory statement

Each student is responsible for complying with rules and regulations that relate to examinations and to academic work in general. The purpose of the mandatory statement is to make students aware of their responsibility and the consequences of cheating. Failure to complete the statement does not excuse students from their responsibility.

*Please complete the mandatory statement by placing a mark **in each box** for statements 1-6 below.*		
1.	**I/we hereby declare that my/our paper/assignment is my/our own work, and that I/we have not used other sources or received other help than mentioned in the paper/assignment.**	☒
2.	**I/we hereby declare that this paper** 1. Has not been used in any other exam at another department/university/university college 2. Is not referring to the work of others without acknowledgement 3. Is not referring to my/our previous work without acknowledgement 4. Has acknowledged all sources of literature in the text and in the list of references 5. Is not a copy, duplicate or transcript of other work	Mark each box: 1. ☒ 2. ☒ 3. ☒ 4. ☒ 5. ☒
3.	I am/we are aware **that any breach of the above will be considered as cheating, and may result in annulment of the examination and exclusion from all universities and university colleges in Norway for up to one year, according to the Act relating to Norwegian Universities and University Colleges, section 4-7 and 4-8 and Examination regulations** section 14 and 15.	☒
4.	**I am/we are aware that all papers/assignments may be checked for plagiarism by a software assisted plagiarism check**	☒
5.	**I am/we are aware that Molde University College will handle all cases of suspected cheating according to prevailing guidelines.**	☒
6.	**I/we are aware of the University College's rules and regulation for using sources**	☒

Personal protection

Personal Data Act

Research projects that processes personal data according to Personal Data Act, should be notified to Data Protection Services (NSD) for consideration.

Have the research project been considered by NSD? □**yes** ☒**no**

- If yes:

Reference number:

- If no:

I/we hereby declare that the thesis does not contain personal data according to Personal Data Act.: ☒

Act on Medical and Health Research

If the research project is effected by the regulations decided in Act on Medical and Health Research (the Health Research Act), it must be approved in advance by the Regional Committee for Medical and Health Research Ethic (REK) in your region.

Has the research project been considered by REK? □**yes** ☒**no**

- If yes:

Reference number:

Publication agreement

ECTS credits: 30

Supervisor: Lars Magnus Hvattum

Agreement on electronic publication of master thesis

Author(s) have copyright to the thesis, including the exclusive right to publish the document (The Copyright Act §2).

All theses fulfilling the requirements will be registered and published in Brage HiM, with the approval of the author(s).

Theses with a confidentiality agreement will not be published.

I/we hereby give Molde University College the right to, free of charge, make the thesis available for electronic publication: ☒yes ☐no

Is there an agreement of confidentiality? ☐yes ☒no

(A supplementary confidentiality agreement must be filled in)

- If yes:

Can the thesis be online published when the period of confidentiality is expired? ☐yes ☐no

Date: 23.05.2022

Preface

This thesis is written as a part of the degree in the Master of Science in Logistics program at Molde University College.

The Institute of Energy Technology (IFE) aims to provide decision support for one of the optimization problems they are currently investigating, where their main goal is to apply heuristic optimization techniques to find good solutions within short computation times, for path finding in environments exposed to radiation. Based on this problem, the case of the thesis is proposed by IFE and the importance of radiation protection.

First of all, we want to thank our supervisor Professor Lars M. Hvattum, who provided us with invaluable support in completing this work, supported us with advice throughout the work, and pointed out unobvious shortcomings in the work. He has made a great contribution to this project and without him we would not have been able to do the work at this level.

This work would not have been possible without the support of the Institute for Energy Technology we are grateful for providing such an exciting and difficult topic for the thesis, as well as for providing the VRdose package, without it this work could not be so plausible.

Denis thanks his friends Mikhail and Pavel for their invaluable support throughout the semester.

Rebecca thanks her cats, family, and friends for the support.

Denis Senokosov Rebecca Nerland

Abstract

Nuclear Power is an important energy source in the world and in need of technological development.

The background of this thesis is a planning problem regarding path finding in radiation contaminated indoor environments, with potentially conflicting objectives. The goal is to make the path planning process less time-consuming and more manageable by creating a support tool for the decision makers that deals with radiation protection challenges.

The work presents three pathfinding algorithms modified for efficient path planning in radiation contaminated area. Two of them are well-known Dijkstra algorithm and A-star algorithm, and the third is a Jump Point Search algorithm.

In this thesis, we have developed the RadPathFinder program for efficient path planning taking radiation into account. This software allows the user to make various experiments with already known algorithms and a modified version of the Jump Point Search algorithm, which is our major contribution from the algorithm perspective. Our modified Jump Point Search method consumes less computational time compared to well-known algorithms, such as A-star and Dijkstra, and finds solutions with similar results.

As far as the authors know, this is the first implementation of the JPS for this particular problem in the literature.

Contents

1. Introduction

Nuclear energy is a vital source of energy for many countries throughout the world. Nuclear technology uses the energy released by splitting the atoms of certain elements. This technology was first developed in the 1940s. During Second World War The nuclear research during the Second World War initially focused on producing bombs. In the 1950s the attention changed to peaceful use of nuclear fission, controlling it for power generation. Today, around 10 % of all energy in the world is supported by nuclear power. The world's nuclear energy is produced by 440 operational nuclear power plants divided amongst 32 countries worldwide (WNA, 2022).

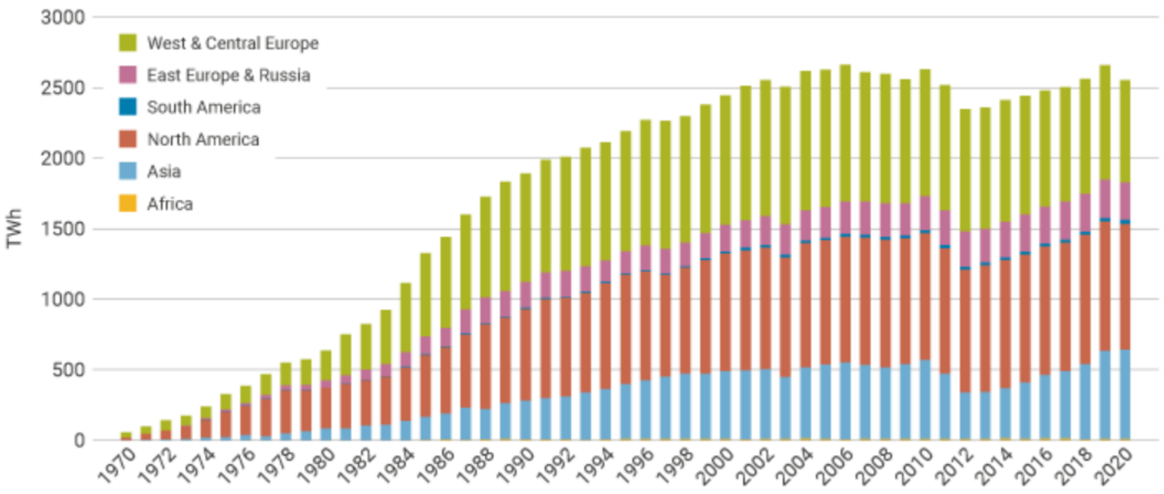


Figure 1.1 Nuclear electricity production from 1970 - 2020 (source: World Nuclear Association, 2022)

From Figure 1.1 we can see that prior to 2020, the electricity generation from nuclear energy had increased for seven consecutive years before 2020. There is a clear need for generating more power capacity around the world, not only to meet increased demand for electricity in many countries, but also to replace old fossil fuel units. The fossil fuel contribution to power generation has remained virtually unchanged the last 10 years or so. In 2018, 64 % of the electricity was generated from burning of fossil fuel. Whereas in 2005, 66.5 % of the power generation came from fossil fuel, despite the fact of the strong support for using renewable electricity sources the recent years (WNA, 2021).

From Figure 1.2 below we can see that in 2019, 10.3 % of the world electricity production came from nuclear power.

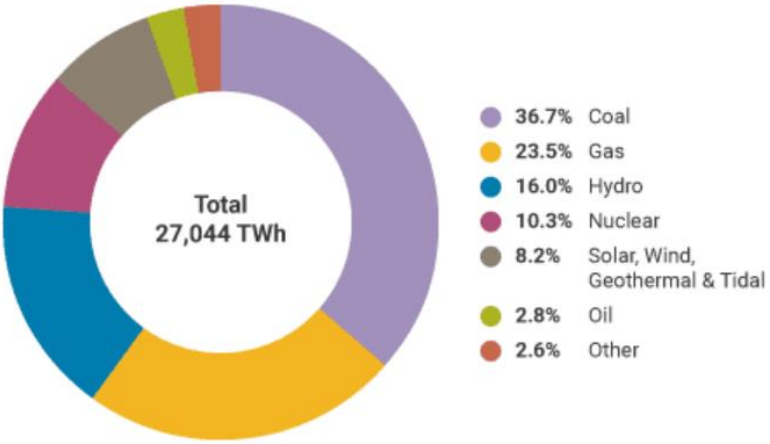


Figure 1.2 World electricity production by source 2019 (source: World Nuclear Association, 2022)

World Nuclear Association (2020) states that the different uses of nuclear technology extend well beyond the possibilities of low-carbon energy. Nuclear technology can help control spread of diseases, assists doctors in their diagnosis and treatments of patients, and powers our most ambitious missions to explore space, to mention some. Due to the multiple and varied uses, nuclear technology is positioned at the heart of the world’s efforts to achieve sustainable developments.

To accomplish more use of nuclear energy, research within the nuclear industry is needed. Radiation protection is one of the focus points for nuclear research today. For the people working at nuclear power plants, the working environment includes high risk and high radiation. It is therefore necessary to take measures to make sure that the staff working in a nuclear power plant are safe. We cannot rely on our senses to avoid radiation because it is not noticeable in smaller doses. A specially constructed route between one place and another is therefore needed to ensure a safe and efficient *path*. Efficient path-planning programs and simulation tools are needed to improve the processes involved and for increasing the safety for people exposed to such environments (Liu et al., 2015).

In the thesis we are focusing on one of the problems The Institute for Energy Technology (IFE) is currently investigating, path finding in environments with radiation exposure. This involves handling multiple objectives, that might be conflicting. The aim of the thesis is to develop a method to find safe and efficient paths to use in nuclear plants. The time it takes to find these paths is also a critical factor, it is therefore important to find efficient solutions within an appropriate time.

The result of the research should be a tool with an optimized algorithm implemented according to the chosen objectives. The algorithms will then be implemented for two-dimensional space. The master's thesis is based on the development of a platform eligible to implement and test different algorithms for the path search in a room exposed with radiation. The platform and programming language used in the thesis is Java. IFE works mainly with Java, and the radiation simulation module is written in the same language. The tests instances will be constructed with the usage of graphical packages. The floor plans of real nuclear power stations are confidential meaning we don't have access to them. The tests instances will therefore be constructed by us with the usage of graphical packages, in our case, with JavaFX.

The growing need of more energy capacity in the world is a clear reason that research must be done to get better decision support regarding path finding.

The thesis is structured as follows. Chapter 2 introduces the Nuclear Industry and energy needs in the world today. Chapter 3 describes the problem background and presents the problem researched in this thesis. Relevant theory and literature are presented in Chapter 4. Chapter 5 gives an insight to the methodology before the implementation details are described in Chapter 6. Chapter 7 presents the findings. Lastly, a conclusion in Chapter 8 and suggestions for further research in Chapter 9.

2. Nuclear Power

This chapter provides background information about the nuclear energy industry and its importance to the world. The importance of the industry in the future, to cover the growing energy needs and to replace the energy production from fossil fuels, is also emphasized. Functional and efficient decision support tools are therefore very much needed to be able to exploit the multiple uses of nuclear power.

Section 2.1 gives an overview of world energy needs, for today and for the future. And the Institute for Energy Technology (IFE) is briefly presented in Section 2.2.

2.1 World Energy Needs

In 2020 thirteen countries produced at least one-quarter of their electricity from nuclear power. France gets approximately three-quarters of its electricity from nuclear energy, Slovakia and Ukraine get more than half from nuclear. Hungary, Belgium, Slovenia, Bulgaria, Finland, and Czech Republic get one-third or more of their electricity from nuclear, while the UK, Spain, Romania, and Russia gets about one-fifth of their electricity from nuclear. Outside Europe we find that South Korea normally gets more than 30 % of its electricity from nuclear. In the USA about one-fifth comes from nuclear. Japan used to rely on nuclear power for more than one-quarter of its electricity and is expected to return to somewhere near that level. Through regional transmission grids, many more countries depend partly on nuclear-generated power. For example, Italy and Denmark get almost 10 % of their electricity from imported nuclear power (WNA, 2022).

In the future the world will need significant increased energy supply, especially cleanly-generated electricity for reducing pollution that effects the climate. Reports on future energy suggest an increasing role for nuclear power as an environmentally benign way of producing reliable electricity on a large scale (WNA, 2021).

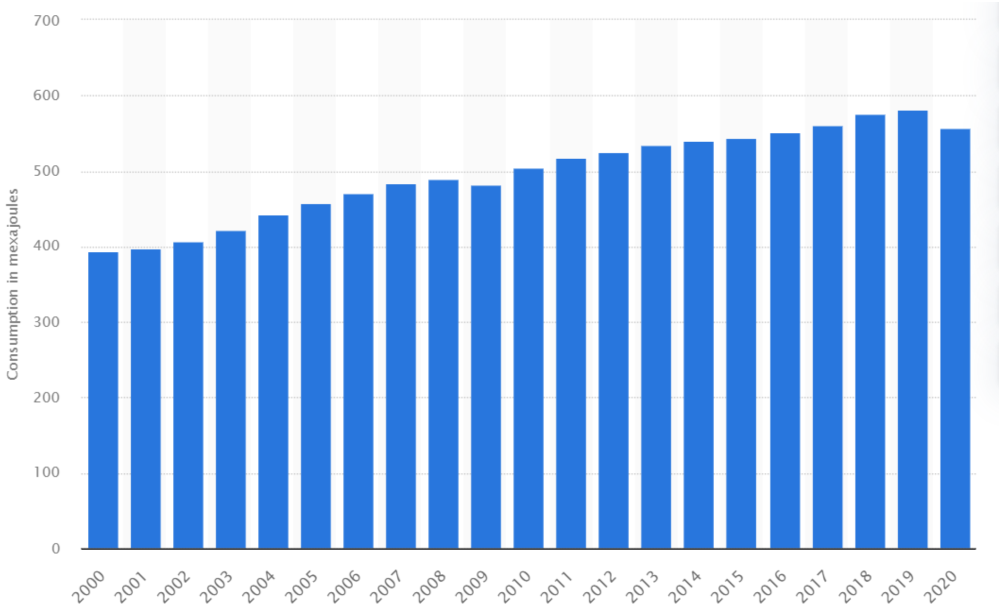


Figure 2.1 Primary energy consumption worldwide from 2000-2020 (source: Statista, 2022)

The energy consumption worldwide has almost had a constant growth in consumption since year 2000. In 2020, due to the response to the coronavirus pandemic, primary energy demand dropped by nearly 4 % as shown in the Figure 2.1 above. The United Nations (UN) estimates that the world's population will grow from 7.6 billion in 2017 to 9.7 billion by 2050. This growth in population alongside the growing economy and rapid urbanization will result in a substantial increase in energy demand over the coming years. As the years go, electrification of end-uses – such as transport, space cooling, large appliances, Information and Communication Technology (ICT) and others are increasing. As a result of this the electricity demand is increasing about twice as fast compared to overall energy use (WNA, 2021).

Today over 80 % of primary energy consumption is from the burning of oil, gas, and coal. Emissions from the combustion of fuels are causing climate change, environmental damage, and the premature death of an estimated 7 million people a year. The key question is therefore: How to reduce harmful emissions, while providing more energy to more people? This positions the energy sector at the heart of achieving sustainable development.

Nuclear power is the only proven, scalable and reliable low-carbon source of energy, and will be required to play a pivotal role if the world is to reduce the use of fossil fuels. By expanding the use of nuclear, modern, and affordable energy can be provided, while reducing the human impact on the natural environment, and ensuring the worlds ability to meet its other sustainable development goals (WNA, 2021).

2.2 The Institute for Energy Technology (IFE)

In the thesis we are assisting IFE with one of the problems they currently are investigating: path finding in environments with radiation exposure. IFE conducts research within the energy industry, this includes nuclear technology, renewable energy, safety and security, oil, and gas to mention some. IFE is located at Kjeller and Halden in Norway, and since 1948 they have been a frontrunner in international energy research. They have contributed to the development of ground-breaking cancer medicine, new solutions in renewable energy, more energy-efficient industrial processes, zero-emission transport solutions and future oriented energy systems (IFE, 2022).

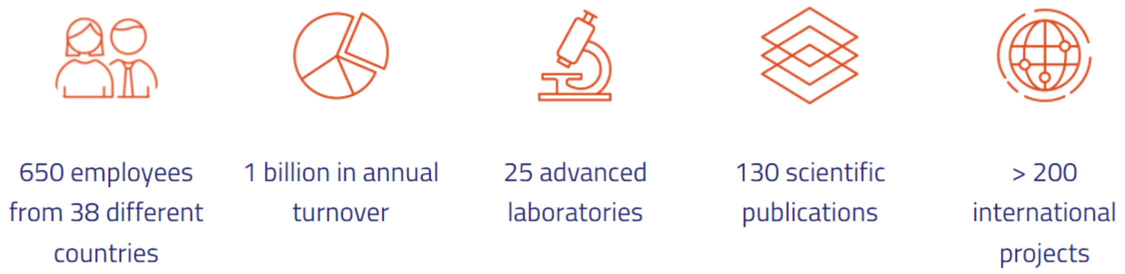

650 employees from 38 different countries
1 billion in annual turnover
25 advanced laboratories
130 scientific publications
> 200 international projects

Figure 2.2 Key figures about IFE (source: IFE, 2022)

IFE's research has led to better nuclear safety in neighbouring countries and around the world. IFE has together with Norwegian industry planned, build and driven 4 nuclear research reactors:

- JEEP I (1951 – 1966)
- The Halden-reactor (1958 – 2018)
- NORA (1961 – 1968)
- JEEP II (1967 – 2019) (Hofstad, K., 2019)

The reactors operations were terminated in 2018-2019, leaving IFE's nuclear activities in a transition phase, preparing for decommissioning. With extensive infrastructure and full-scale laboratories, theoretical models are transformed into commercial activities. Within radiation protection and environmental monitoring of radioactive and chemical emissions IFE has unique expertise and systems in place, making them an important partner for companies that want to research, develop, and produce new solutions for renewable energy and medicine using radioactive sources (IFE, 2022).

The radiation protection departments major responsibilities are personnel dosimetry and radiological monitoring of both the working and the external environments. The department runs employee training programs within the field of radiation protection. For external customers they offer the possibility to measure radioactive content in various products. Further the department uses its knowledge to provide advice and conduct experiments involving the use of radioactive substance (IFE, 2022).

3. Problem Background

The thesis focuses on developing a tool with an efficient path finding algorithm for finding sufficient paths to be used inside buildings with radiation sources. During the last 50 years the world witnessed at least two devastating technogenic catastrophes. In 1986 the reactor number 4 blew up at Chornobyl Nuclear plant and caused the death of thousands of people. In 2011, after the strongest tsunami in Japan's history, Fukushima nuclear plant was flooded. In both cases the personnel and rescuers had to risk their life's and operate in contaminated areas. The importance of having access to an easy tool with an efficient algorithm to find safe paths fast is clear if the usage of nuclear power is to grow in the future. In this chapter details about radiation are presented in Section 3.1 including health effects and regulations. Section 3.2 explains how radiation is simulated in the thesis. The problem description with limitations and assumptions are presented in Section 3.3.

3.1 Radiation

Today, radiation is a well understood process, where most of the radiation we all receive yearly comes from natural sources. As seen in Figure 3.1 below, 85 % of the radiation exposure comes from natural radiation, and only 1 % comes from the nuclear industry. Contrary to the public perception, nuclear power accidents have caused very few fatalities and the use of nuclear energy does no expose members of the public to significant radiation levels (WNA, 2022).

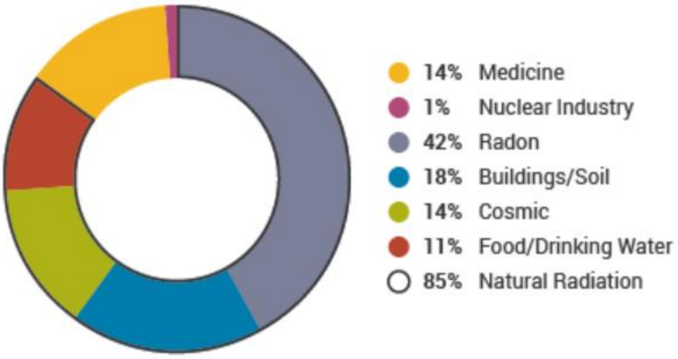


Figure 3.1 Sources of radiation (source: NWA, 2022)

World Nuclear Association (2022) states that in 2016 a report from the United Nations Environment Programme (UNEP) noted: “We know more about the sources and effects of exposure to radiation than to almost any other hazardous agent, and the scientific community is constantly updating and analysing its knowledge… The sources of radiation causing the greatest exposure of the general public are not necessarily those that attract the most attention”.

As a result of naturally occurring radioactive elements in, for example, the soil, the air and the human body, radiation is constantly surrounding us. Radiation associated with nuclear medicine and the use of nuclear energy is “ionizing” radiation, meaning that the radiation has sufficient energy to interact with matter. The interaction between ionizing radiation and living tissue can cause damage to people’s health (WNA, 2022).

3.1.1 Health effects

Ionizing radiation has sufficient energy to affect the atoms in living cells and thereby damage their genetic material (DNA). Fortunately, the cells in our bodies are extremely efficient at repairing this damage. If the damage is not repaired correctly, a cell may die or eventually become cancerous. Exposure to very high levels of radiation, such as being close to an atomic blast, can cause acute health effects such as skin burns and acute “radiation sickness”. Radiation sickness can cause symptoms such as nausea and vomiting within hours after exposure and can sometimes result in death over the following days or weeks. High levels of radiation exposure can also result in long-term health effects such as cancer and cardiovascular disease. Exposure to low levels of radiation encountered in the environment does not cause immediate health effects but is a minor contributor to our overall cancer risk (EPA, 2022).

3.1.2 Units of radiation

The basic unit of radiation dose absorbed in tissue is the gray (Gy), where one gray represents the deposition of one joule of energy per kilogram of tissue. However, since some types of radiation causes more damage per grey than others, another unit sievert (Sv) is used in setting radiological protection standards. This weighted unit of measurement considers biological effects of different types of radiation and indicated the equivalent dose.

Calculated dose for humans is normally measured in millisieverts (mSv), one thousand of a sievert. Comparing the biological effect of *one gray* of the different types of radiation we get Table 3.1 below:

Type of Radiation	**Effect in sievert (Sv)**	**Millisievert (mSv)**
Alpha	20	0.02
Beta	1	0.001
Gamma	1	0.001
Neutrons	10	0.01

Table 3.1 Table of the biological effect of one gray from different types of radiation.

Sievert and gray measurements are accumulated over time, whereas damage depends on the actual dose rate, e.g., mSv per day or year. The average dose received by humans from background radiation is around 2.4 mSv a year. Depending on the geology and altitude where people live, this dose can vary between 1 and 10 mSv per year and can even be more than 50 mSv per year (WNA, 2022).

3.1.3 Limiting radiation exposure

Natural background radiation coming primarily from natural minerals, is around us all the time, making it difficult to limit the exposure to it. In other cases, such as radiation exposure from a nuclear power plant, three measures can be taken to protect people from identified radiation sources:

1. Limiting time
2. Distance
3. Shielding

These three measures can also be seen in Figure 3.2. In occupational situations, dose rate is reduced by limiting or minimizing the exposure time. The intensity of radiation decreases with distance from its source, just like heat from fire reduces as you move further away. Shielding, as barriers of lead, concrete or water provide protection from penetrating radiation, such as gamma rays. Inserting the proper shield between you and a radiation source will greatly reduce or eliminate the dose you receive (EPA, 2022).

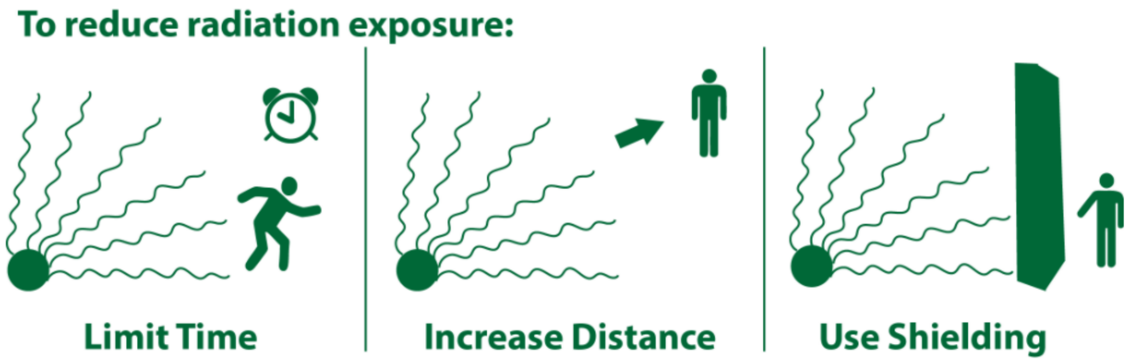


Figure 3.2 Ways to limit radiation exposure (source: EPA, 2022)

3.1.4 Regulations

In any country, radiation protection standards are set by the government authorities, generally in line with recommendations by the International Commission on Radiological Protection (ICRP). The three key points of the ICRP's recommendations are:

- *Justification:* No practice should be adopted unless its introduction produces a positive net benefit.
- *Optimization:* All exposures should be kept as low as reasonably achievable (ALARA), economic and social factors being taken into account.
- *Limitation:* The exposure of individuals should not exceed the limits recommended for the appropriate circumstances.

National radiation protection standards are framed for the Occupational exposure category as follows: The maximum permissible dose for occupational exposure should be 20 mSv per year, averaged over 5 years. This gives a total of maximum 100 mSv in 5 years, with a maximum of 50 mSv in any one year. The figures are over and above natural background levels, and exclude medical exposure (WNA, 2022).

3.2 Radiation Simulation

To simulate radiation environments, IFE has been involved in research and development of applications for computer simulation and virtual reality (VR) technology. Two significant software tools have been developed, the VRdose system and the Halden Planner (Szöke et al., 2014).

The HVRC VRdose is a real-time software tool for modelling and characterising nuclear environments, planning a sequence of activities in the modelled environment, optimising protection against radiation, and producing job plan reports with dose estimates. A significant motivation behind the VRdose was to make advanced, radiation exposure situation analysis technology accessible to border range of users, who can benefit from an interactive 3D visual representation of radiation risks to support ALARA optimization (IFE, 2020). Independent development of a tool for radiation simulation is a very challenging task that requires a considerably large amount of time. For that reason, IFE has shared with us the radiation simulation software from their commercial product the HVRC VRdose.

3.3 Problem Description

The purpose of this section is to present the thesis problem, including assumptions and limitations. The aim of the thesis is to assist IFE by creating an algorithm, that performs better than what they previously have used for efficient path planning within buildings with radiation sources. In other words, the new algorithm should have a low computational time and the solutions found should not be far from an optimal path. A platform for creating instances, radiation simulation, and testing selected algorithms is needed to solve this problem. The solutions found with the algorithms implemented in the platform, can be used as decision support to assist in solving challenges faced with path planning in nuclear plants.

In the thesis we are studying a problem that concerns finding optimized paths in different room structures inside a building exposed with radiation. We consider two possible ways of movement. The first is getting from a starting point directly to an end point and the second is getting from a starting point then visit several other points in between before returning to the starting point again.

The problem can therefore be seen as *the shortest path problem* and *the travelling salesman problem*, which are well known and studied path planning problems. These specific problems are described later in Chapter 4.

When walking around inside buildings we often encounter obstacles that we can't move through, for example a wall. This means that we must find another way, besides moving directly in a straight line, to get to where we want to be. We also don't want to waste time walking in circles, meaning we desire to find the shortest way there. Another concern is the radiation sources that can be found inside the buildings. The dose rate on different points in a room are different depending on where the radiation sources are located. For this problem a software must be developed for implementation of path-planning algorithms, to simulate radiation sources, and for graphical layout of the space that needs to be navigated through. From Figure 3.3 below, we can see a simple map of the Lincoln Park Zoo, which is divided into squares numbered from 1 to 4 and lettered from A to G, making it a *grid map*.

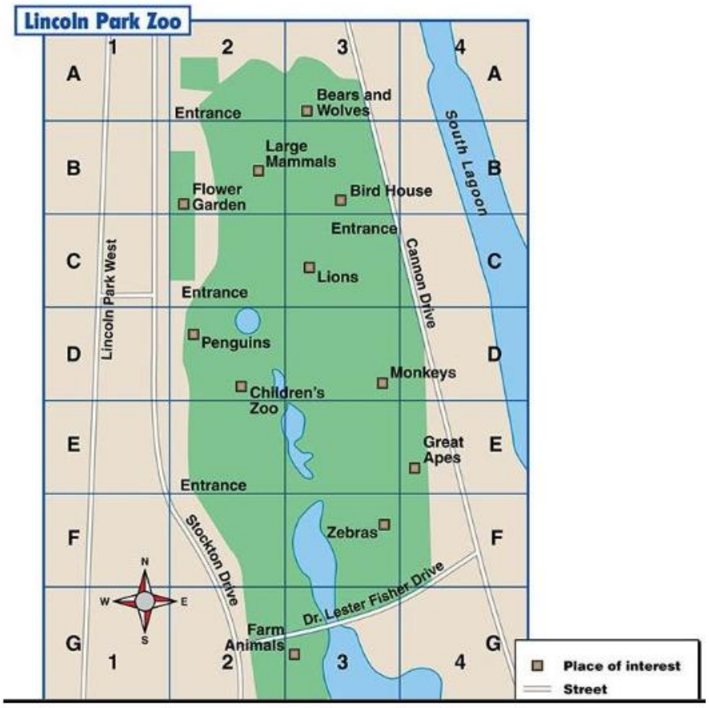


Figure 3.3 Grid map of the Lincoln Park Zoo (source: Hilinski, J.).

Based on this map the Shortest Path Problem can be seen as getting from point Farm Animals to Great Apes. Because of the lake between these two points, you would either have to walk around the lake or follow the street over the lake. From the Traveling Salesman Problem point of view a start point could be the Children's Zoo, visiting all the other places of interest in between and returning to the Children's Zoo again.

With a suitable tool, the shortest path for both these problems can be found. For solving shortest path problems, the algorithm searches for possible paths with a rule saying that minimum distance leads the search. This algorithm can be modified so that the search is led by another rule, as for example minimum radiation dose.

Two goals are chosen to be the focus of this thesis problem. The first goal is to minimize the path length. This is going to reduce the time it takes to get to the final point and the time exposed to radiation. The second goal is to minimize the dose rate, providing safety for people exposed to radiation. Satisfying this goal is important when it comes to the development of nuclear energy in the future. On the other end, this goal overlooks the challenges related to efficiency when it comes to using the one with the lowest dose rate instead of a shorter one. These two goals can be conflicting and non-conflicting depending on what is the shortest path and where the dose rate is at its lowest or highest.

Finding a way to efficiently solve these goals can assist decision makers in efficient path planning. One important criterium for the new algorithm is the computational time, meaning that we would like a fast algorithm that can provide solutions quickly. The developed tool for this problem should include suitable implemented algorithms to find good solutions and making it possible to test several algorithms on the same instance so the decision maker can select the preferred solution of these.

3.3.1 Assumptions and limitations

Instances

Nuclear plant maps are confidential, which means we are not able test the algorithms on instances from real nuclear plants. Instances used for testing different algorithms are therefore created by us for testing purposes. All the maps are constructed in two dimensional.

Path

The solutions found, should be to a feasible way to get from the start state to the goal state. Feasible means that it is possible to use the solution found. These solutions we will refer to as paths when fitted.

Dose rate

The radiation doses of each point on a map are computed by VRdose software. The doses are precomputed once an instance is created. Accumulated doses of paths are computed with the usage of constant speed for the entire path.

Obstacles

We assume that obstacles in the building/rooms are static and known, meaning that no obstacles would move, and full information about the map including the obstacles are provided in advance.

Grid

The maps are presented as grid maps with a constant size of edges equal to 0.1 meters. In path planning, the algorithms do not consider the physical size or manoeuvring characteristics of an object traveling found a path. Therefore, a small gap of 0.1 meters between two objects is considered traversable.

4. Related Theory and Literature

This chapter contains an overview over relevant theory and literature about optimization and planning problems regarding path finding, and methods purposed from the literature to solve multi-objective optimization problems. This specific area is chosen to get an overview of what has previously been done, and what this thesis can contribute with.

The problem studied in this thesis can be seen as a shortest path problem where someone needs to get from one point to another, or if someone needs to visit several points before returning to the starting point, as quickly as possible by minimizing the distance. Based on our knowledge, the problem studied in this report has never been studied before. This being the case, there is only supporting literature available, although the shortest path problem has been studied before. However, those problems are not considering radiation sources or being inside a building.

Section 4.1 gives an overview of the characteristics of graph theory, showing how our surroundings can be presented mathematically, making it possible to solve such problems with programming. In Section 4.2, grid maps are presented as an extension to graph theory. Section 4.3 describes possible ways to measure distance. The two different path planning problems are described in Section 4.4 and Section 4.5. Selected path algorithms for solving these problems, which can be modified for our own purpose, are presented in Section 4.6. Section 4.7 describes the multi-objective optimization problem including different ways to perform multi-objective optimization. The chosen solution methods from the literature are presented in Section 4.8.

4.1 Graph Theory

Many real-world situations can easily be described by diagrams consisting of a set of points together with lines, joining certain pairs of these points. The points could for example, represent people, with lines joining pairs of friends. The points can also be seen as communication centres, with the lines representing communication links. In such diagrams one is mainly interested in whether two given points are joined by a line, how they are joined is immaterial. A mathematical abstraction of situations of this type gives rise to the concept of a graph (Bondy & Murty, 2008).

Bondy & Murty (2008) defines a *graph* G as an ordered pair $\big(V(G), E(G)\big)$ consisting of a set $V(G)$ of *vertices* and a set $E(G)$, of edges, together with an incidence function ψ_G that associates with each edge of G an unordered pair of vertices of G. If e is an edge and u and v are vertices such that $\psi_G(e) = \{u, v\}$, then e is said to *join* u and v, and vertices u and v are called the ends of e. We denote the number of vertices and edges in G by $(v)G$ and $e(G)$, these two basic parameters are called the *order* and *size* of G, respectively. Two examples of graphs are presented to clarify the definition. For notational simplicity, we write uv for the unordered pair $\{u, v\}$.

Example 1.

$$G = \big(V(G), E(G)\big)$$

where

$$V(G) = \{u, v, w, x, y\}$$

$$E(G) = \{a, b, c, d, e, f, g, h\}$$

and ψ_G is defined by

$$\psi_G(a) = uv \quad \psi_G(b) = uu \quad \psi_G(c) = vw \quad \psi_G(d) = wx$$

$$\psi_G(e) = vx \quad \psi_G(f) = wx \quad \psi_G(g) = ux \quad \psi_G(h) = xy$$

Example 2.

$$H = (V(H), E(H))$$

where

$$V(H) = \{v_0, v_1, v_2, v_3, v_4, v_5\}$$
$$E(H) = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8, e_9, e_{10}\}$$

and ψ_H is defined by

$$\psi_H(e_1) = v_1v_2 \quad \psi_H(e_2) = v_2v_3 \quad \psi_H(e_3) = v_3v_4 \quad \psi_H(e_4) = v_4v_5 \quad \psi_H(e_5) = v_5v_1$$
$$\psi_H(e_6) = v_0v_1 \quad \psi_H(e_7) = v_0v_2 \quad \psi_H(e_8) = v_0v_3 \quad \psi_H(e_9) = v_0v_4 \quad \psi_H(e_{10}) = v_0v_5$$

The name *graphs* come from the fact that they can be represented graphically, and it is this graphical representation that helps us understand many of their properties. Each vertex is indicated by a point, and each edge by a line joining the points representing ends. Diagrams of G and H are shown in Figure 4.1, where the vertices are represented by small circles.

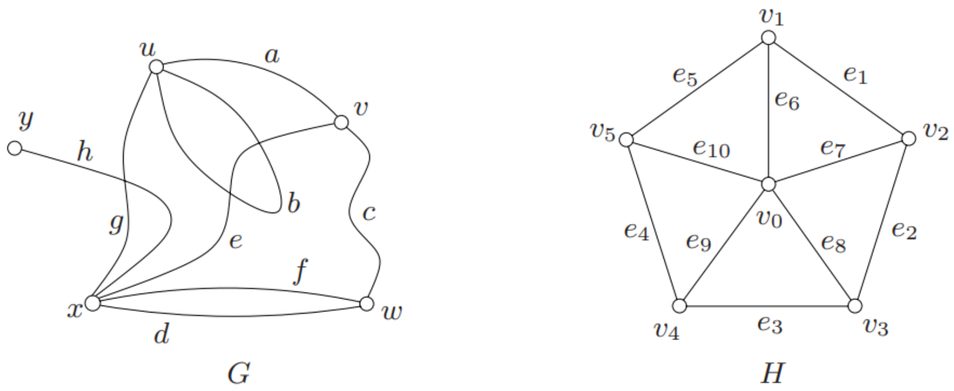


Figure 4.1 Diagrams of the graphs *G* and *H* (source: Bondy & Murty, 2008).

Most of the definitions and concepts in graph theory are suggested by this graphical representation. The ends of an edge are said to be incident with the edge, and vice versa. Two vertices which are incident with a common edge are adjacent, as are two edges which are incident with a common vertex, and two distinct adjacent vertices are neighbors. The set of neighbors of a vertex v in a graph G is denoted by $N_G(v)$. A *loop* is an edge with two identical ends, and an edge with two distinct ends is called a *link*. Two or more links with the same pair of ends are said to be *parallel edges*. From Figure 4.1 above we can see that in graph *G*, the edge *b* is a loop, and all other edges are links, in addition edges *d* and *f* are parallel edges (Bondy & Murty, 2008).

A graph is *simple* if it has no loops or parallel edges. The graph H in Example 2 is simple, while the graph G in Example 1 is not. Much of graph theory is concerned with the study of simple graphs. A set V, together with a set E of two-element subsets of V, defines a simple graph (V, E), where the ends of an edge uv are precisely the vertices u and v. In any simple graph we can dispense with the incidence function ψ by renaming each edge as the unordered pair of its ends. In a diagram of such a graph, the labels of edges may then be omitted (Bondy & Murty, 2008).

A *path* is a simple graph whose vertices can be arranged in a linear sequence in such a way that two vertices are adjacent if they consecutive in the sequence and are nonadjacent otherwise as seen in Figure 4.3(a) below. A cycle on three or more vertices is a simple graph whose vertices can be arranged in a cyclic sequence, such that two vertices are adjacent if they are consecutive in the sequence and are nonadjacent otherwise as seen from Figure 4.3(b) below. The *length* of a path or a cycle is the number of its edges. A path or cycle of length k is called a *k-path* or *k-cycle,* the path or cycle is odd or even according to the parity of k. If the path or cycle contains every vertex of a graph it is called a Hamilton path or Hamilton cycle of the graph. A graph is *traceable* if it contains a Hamilton path, and *Hamiltonian* if it contains a Hamilton cycle (Bondy & Murty, 2008).

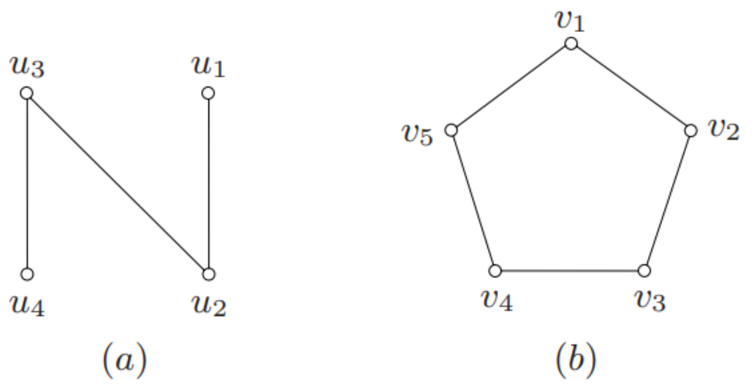
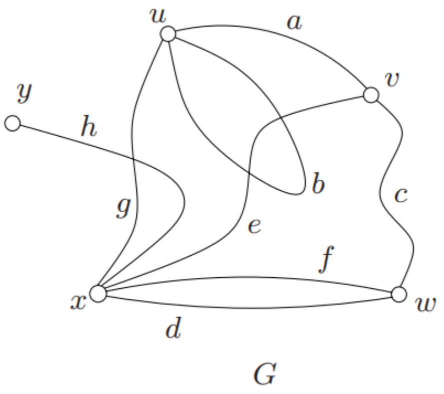


Figure 4.3 (a) A path of length three, and (b) a cycle of length five (source: Bondy & Murty, 2008).

4.1.1 Incidence and Adjacency Matrices

Even though drawings are convenient for specifying graphs, they are not suitable for storing graphs in computers, or for applying mathematical methods to their properties. For these purposes, we consider two matrices associated with a graph, its incidence matrix and its adjacency matrix. Let G be a graph, with vertex set V and edge set E. The *incidence matrix* of G is the $n \times m$ matrix $\boldsymbol{M}_G := (m_{ve})$, where m_{ve} is the number of times (0, 1 or 2) that vertex v and edge e are incident. The incidence matric is another way of specifying the graph. The *adjacency matrix* of G is the $n \times n$ matrix $\boldsymbol{A}_G := (a_{uv})$, where a_{uv} is the number of edges joining vertices u and v, each loop counting as two edges. When dealing with simple graphs, and even more compact representation than the adjacency matrix is possible. For each vertex v, the neighbours of v are listed in some order. A list $(N(v) : v \in V)$ of these lists is called an *adjacency list* of the graph. Simple graphs are usually stored in computers as adjacency lists (Bondy & Murty, 2008). Figure 4.4 below shows the incidence metric and adjacency metric of graph G from Example 1 shown previous in this chapter.



	a	b	c	d	e	f	g	h
u	1	2	0	0	0	0	1	0
v	1	0	1	0	1	0	0	0
w	0	0	1	1	0	1	0	0
x	0	0	0	1	1	1	1	1
y	0	0	0	0	0	0	0	1

M

	u	v	w	x	y
u	2	1	0	1	0
v	1	0	1	1	0
w	0	1	0	2	0
x	1	1	2	0	1
y	0	0	0	1	0

A

Figure 4.4 Incidence (M) and adjacency (A) matrices of graph G (source: Bondy & Murty, 2008).

4.1.2 Vertex Degrees

The *degree* of a vertex v in a graph G, denoted by $d_G(v)$, is the number of edges of G incident with v, each loop counting as two edges. If G is a simple graph, $d_G(v)$ is the number of neighbors of v in G. A vertex of degree zero is called an *isolated vertex*. For graph G, we denote the minimum degrees of vertices by $\delta\ (G)$ and the maximum degrees of vertices by $\Delta(G)$. The *average degree* is denoted by $d(G)$ and calculated by $\frac{1}{n}\sum_{v\in V} d(v)$.

The following theorem establishes a fundamental identity relating the degrees of the vertices of a graph and the number of its edges (Bondy & Murty, 2008).

Theorem 4.1 *For any graph G,*

$$\sum_{v \in V} d(v) = 2m \qquad (1)$$

4.2 Grid Maps

Path planning on grid graphs has been studied for decades and is widely applied in areas such as robotics and computer science. The grid is a simple approach that discretizes continuous spaces and represents maps as graphs. Further, girds are used in complex obstacle fields owing their simplicity and certainty in terms of the location of vertices and edges. In general, free space and obstacles are represented as unblocked and blocked cells, respectively, in grid graphs. Each cell serves as a vertex in a graph, and edges are generally restricted from each cell to its adjacent neighbours (typically, eight neighbours). It aims to find a short, unblocked path from a given start vell and to a given goal cell where an agent can always move from its current cell to any one of it unblocked neighbouring cells (Han & Koenig, 2021).

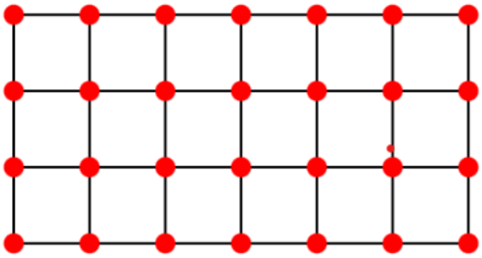


Figure 4.5 Example of a grid graph (source: Weisstein, E. W.)

A two-dimensional grid graph, also known as a rectangular grid graph or two-dimensional lattice graph, is an $m \times n$ lattice graph that is the graph Cartesian product $P_m \times P_n$ of path graphs on m and n vertices. The Cartesian graph product $G = G_1 \times G_2$ of graphs G_1 and G_2 with disjoint point sets V_1 and V_2 and edge sets X_1 and X_2 is the graph with point set $V_1 \times V_2$ and $u = (u_{1,} u_2)$ adjacent with $v = (v_1, v_2)$ whenever $[u_1 = v_1$ and u_2 adj $v_2]$ or $[u_2 = v_2$ and u_1 adj $v_1]$ (Weisstein, E. W., 2022).

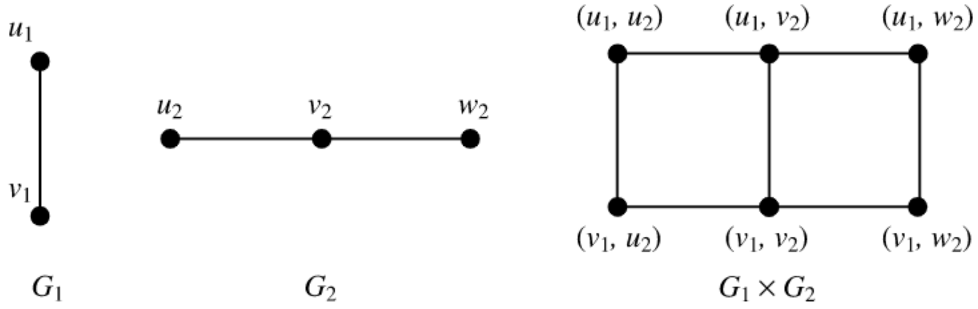


Figure 4.6 The Cartesian Product (source: Weisstein, E. W.)

4.3 Distance Metrics

When using grid maps for path planning the distance between two vertices or points must be computed. Euclidean distance and Manhattan distance are well known distance metrics which compute a number based on two points.

Euclidean distance

Euclidean distance is the shortest distance between two points in an N dimensional space, also known as Euclidean space. The Euclidean distance in two-dimensional space is calculated by the given Equation (1) below. A point is represented as (x, y).

$$\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} \tag{1}$$

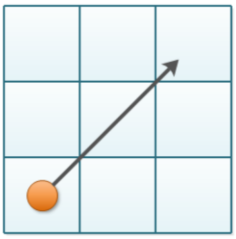


Figure 4.7 The concept of Euclidean distance (source: Chatterjee, A.)

Properties of Euclidean distance is that there is an unique path between two points whose length is equal to Euclidean distance. For a given point, the other point lies in a circle such that the Euclidean distance is fixed. The radius of the circle is the fixed Euclidean distance, this is shown in the figure below (Chatterjee, A., 2022).

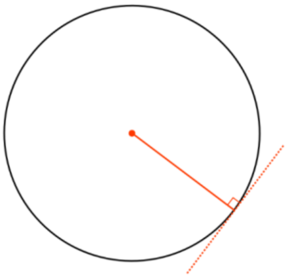


Figure 4.8 Fixed Euclidean Distance (source: Chatterjee, A.)

Manhattan Distance

Manhattan distance is a distance metric between two points in a N dimensional vector space. It is the sum of the lengths of the projections of the line segment between the points onto the coordinate axes. In simple terms, it is the sum of absolute difference between the measures in all dimensions of two points. In a two-dimensional space, the Manhattan distance between two points is given by Equation 2 below (Chatterjee, A., 2022):

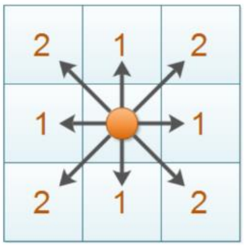


Figure 4.9 The concept of Manhattan distance (source: Chatterjee, A.)

$$|x_1 - x_2| + |y_1 - y_2| \qquad (2)$$

4.4 The Shortest Path Problem

Shortest path problems are alluring to both researchers and to practitioners for several reasons. One of them are that they arise frequently in practice since in a wide variety of application settings we wish to send some material (e.g., a computer data packet, a telephone call, a vehicle) between two specified points, as quickly, as cheaply, or as reliably as possible (Ahuja et al., 1993).

We consider a directed graph $G = (V, E)$ with an edge length (or edge cost) c_{ij} associated with each edge $(i, j) \in E$. The graph has a distinguished vertex s, called the source. Let $E(i)$ represent the arc adjacency list of vertex i and let $C = \max \{c_{ij} : (i, j) \in E\}$. We define the *length of a directed path* as the sum of lengths of edges in the path. The shortest path problem is to determine for every non-source vertex $i \in V$ a shortest length directed path from vertex s to vertex i. Alternatively, we might view the problem as sending one unit of something as cheaply as possible (with edge unit cost as c_{ij}) from vertex s to each of the vertices in $V - \{s\}$ in an incapacitated network. This viewpoint gives rise to the following linear programming formulation of the shortest path problem:

$$Minimize \sum_{(i,j)\in E} c_{ij}x_{ij} \qquad (3)$$

Subject to:

$$\sum_{\{j:(i,j)\in E\}} x_{ij} - \sum_{\{j:(j,i)\in E\}} x_{ji} = \begin{cases} n-1 & for\ i = s \\ -1 & for\ all\ i \in V - \{s\} \end{cases} \qquad (4)$$

$$x_{ij} \geq 0 \quad for\ all\ (i,j) \in E \qquad (5)$$

For the shortest path problem, we impose several assumptions, the first being that all edge lengths are integers. This assumption imposed on edge lengths is necessary for some algorithms and unnecessary for others. Algorithms whose complexity bound depends on C assume integrality of the data. We can also transform rational edge lengths to integer edge lengths by multiplying them by a suitable large number. Moreover, we necessarily need to convert irrational numbers to rational numbers to represent them on a computer. Therefore, the integrality assumption is not a restrictive assumption in practice.

The second assumption is that the network contains a directed path from every vertex s to every other vertex in the network. The third is that the network does not contain a negative cycle, and the last is that the network is directed (Ahuja et al., 1993).

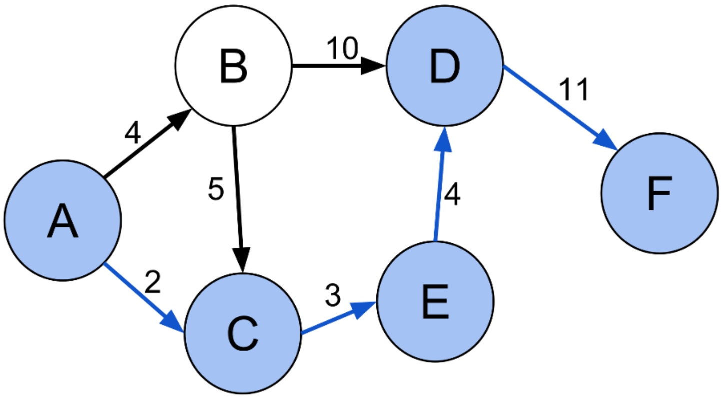


Figure 4.10 Shortest path (A, C, E, D, F) between vertices A and F in the weighted directed graph (source: Wikipedia)

4.5 The Traveling Salesman Problem

The traveling salesman problem (TSP) were studied in the 18th century by a mathematician from Ireland named Sir William Rowman Hamilton and by the British mathematician named Thomas Penyngton Kirkman. Given a set of cities and the cost of travel (or distance) between each possible pairs, the TSP, is to find the best possible way of visiting all the cities and returning to the starting point that minimize the travel cost (or travel distance) (Matai et al., 2010).

Broadly, the TSP is classified as symmetric travelling salesman problem (sTSP), asymmetric traveling salesman problem (aTSP), and multi travelling salesman problem (mTSP). A short description of these three is presented below.

sTSP:

Let $V = \{v_1, \dots \dots, v_n\}$ be a set of cities,

$A = \{(r, s): r, s \ \in V$ be the edge set,

and $d_{rs} = d_{sr}$ be the cost measure associated with edge $(r, s) \in A$.

The sTSP is the problem of finding a minimal length closed tour that visits each city once. In this case cities $v_i \in V$ are given by their coordinates (x_i, y_i) and d_{rs} is the Euclidean distance between r and s then we have an Euclidean TSP.

aTSP:

If the cost measure $d_{rs} \neq d_{sr}$ for at least one (r, s) then the TSP becomes an asymmetric travelling salesman problem.

mTSP:

The mTSP is defined as: In a given set of nodes, there are m salesmen located at a single depot node. The remaining nodes that are to be visited are intermediate nodes. Then, the mTSP consists of finding tours for all m salesmen, who all start and end at the depot, such that each intermediate node is visited exactly once and the total cost of visiting all nodes is minimized. The cost metric can be defined in terms of distance, time, etc. Possible variations of the problem are as follows:

- *Single vs. multiple depots:* In the single depot, all salesmen finish their tours at a single point while in multiple depots the salesmen can either return to their initial depot, or to any depot keeping the initial number of salesmen at each depot the same after the travel.
- *Number of salesmen:* The number of salesmen in the problem can be fixed or a bounded variable.
- *Cost:* When the number of salesmen is not fixed, then each salesman usually has an associated fixed cost incurring whenever this salesman is used. In this case, minimizing the requirements of salesmen also becomes an objective.
- *Timeframe:* Some nodes need to be visited in a particular time periods. These periods are called time windows and is an extension of the mTSP, referred to as multiple traveling salesman problem with specifies timeframe (mTSPTW).
- *Other constraints:* Constraints can be the number of nodes each salesman can visit, maximum or minimum distance a salesman travels or others (Matai et al., 2010).

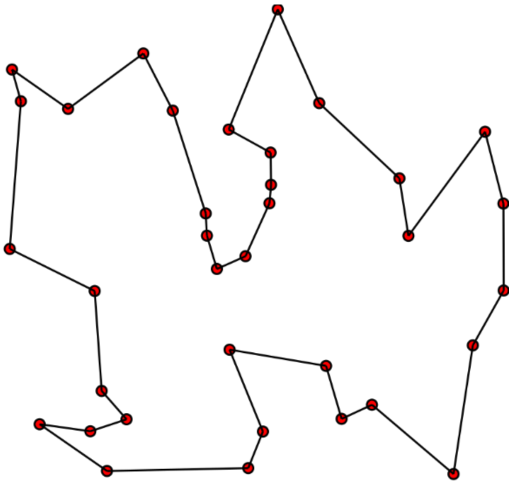


Figure 4.11 Solution of a TSP: the black line shows the shortest possible loop that connects every red dot (source: Wikipedia)

The basic traditional model for the TSP can be stated as an assignment problem with "subtour elimination" constraints. Let n denote the number of cities; x_{ij} is a binary decision variable which is equal to 1 if there is travel from city i to city j, and 0 if not; and c_{ij}, is the cost of travel from city i to city j. The basic traditional model for the TSP is as follows (Diaby & Karwan, 2016):

$$Minimize \sum_{i=1}^{n}\sum_{j=1}^{n} c_{ij}x_{ij} \tag{6}$$

Subject to:

$$\sum_{j=1}^{n} x_{ij} = 1; \;\; i = 1, \dots, n \tag{7}$$

$$\sum_{i=1}^{n} x_{ij} = 1; \;\;\; j = 1, \dots, n \tag{8}$$

$$x_{ij} \;\in \{0,1\}; \;\;\; i, j = 1, \dots, n \tag{9}$$

$$\{(i,j): x_{ij} = 1, \;\; i, j = 2, \dots, n\} \tag{10}$$

4.6 Path Algorithms

In the literature many different types of algorithms are presented to solve path finding problems. Some of the most popular conventional shortest path algorithms are presented in this section.

4.6.1 Dijkstra’s algorithm

Dijkstra’s algorithm finds shortest paths from the source vertex s to all other vertices in a network with non-negative edge lengths. Dijkstra’s algorithm maintains a distance label $d(i)$ with each vertex i, which is an upper bound on the shortest path length to vertex i. At any intermediate step, the algorithm divides the vertices into two groups: 1. those which it designates as permanently labelled (or permanent), and 2. those it designates as temporarily labelled (or temporary). The distance label to any permanent vertex represents the shortest distance from the source to that vertex. For any temporary vertex, the distance label is an upper bound on the shortest path distance to that vertex. The basic idea of the algorithm is to fan out from vertex s and permanently label vertices in the order of their distances from vertex s. Initially we give vertex s a permanent label of zero, and each other vertex j a temporary label equal to ∞. At each iteration, the label of a vertex i is its shortest distance from the source vertex along a path whose internal vertices (i.e., vertices other than s or the vertex i itself) are permanently labelled. The algorithm selects a vertex i with the minimum temporary label, makes it permanent, and reaches out from that vertex – that is, scans edges in $E(i)$ to update the distance labels of adjacent vertices. The algorithm terminates when it has designated all vertices as permanent (Ahuja et al., 1993).

Step 1: Mark the initial vertex (s). Let:
$d(s) = 0$ (Constant distance label)
$d(x) = \infty$ (Tentative distance label)
$p = s$ (p – last marked vertex)

Step 2: (Changing the tentative distance labels).
For all unmarked vertices x the numbers $d(x)$ are recalculated by equation 11:

$$d(x) = \min\ \{d(x), d(p) + c(p, x)\} \tag{11}$$

Where $d(x)$ is the minimal tentative distance from s to $x, d(p)$ – minimal tentative distance from s to p and $c(p, x)$ is the edge's weight from p to x. (Obviously only $d(x)$ for which the edge (p, x) exists may be changed, the rest of the numbers remain the same). If $d(x) = \infty$, for each unmarked vertex x, stop the procedure (it means that there are no paths from s to all unmarked vertices). Otherwise mark the one vertex x which has minimal distance label $d(x)$. Also colour the edge which goes into vertex x, for which the minimal distance label from Eqn. 11 is reached. Let $p = x$.

Step 3: If $p = t$, the procedure ends, the only way from s to t, made from marked edges, is the shortest path between s and t. Otherwise, go back to *Step 2* (Sapundzhi & Popstoilov, 2017).

Dijkstra's algorithm using adjacency matrix

Step 1: All vertices are numbered with the numbers from 1 to n. Matrix $D^0 = (d_{ij}^0)_{nxn}$ is determined. Element (i, j) is the shortest edge's length (with least weight) between i and j. $d_{ij}^0 = \infty$ if (i, j) edge is missing and $d_{ii}^0 = 0, \forall i$.

Step 2: The distance labels $d(x)$ are recalculated for each unmarked vertex x where an edge exists from p to x by Eqn.11: $d(x) = \min\{d(x), d(p) + c(p, x)\}$, where x and p are respectively the distances array's indices of vertices x and current marked vertex. Then it is appropriated p the index of this unmarked vertex x which has minimal distance label and colored the edge entering x for which the minimal number from the formula was reached.

Step 3: If the current vertex index equals the target vertex index $p = t$, the procedure ends. The only path from s to t, made from marked edges, is the shortest path between s and t. Otherwise go back to *Step 2* (Sapundzhi & Popstoilov, 2017).

4.6.2 A-star algorithm

A-star algorithm is one of the best-known path planning algorithms, which can be applied on metric or topological configuration space. This algorithm uses combination of heuristic searching and searching based on the shortest path. The A-star algorithm is classified as a best-first algorithm, because each cell in the configuration space is evaluated by the value:

$$f(v) = h(v) + g(v) \tag{12}$$

Where $h(v)$ is heuristic distance (Manhattan or Euclidean) from the start point to the goal state, and $g(v)$ is the length of the path from the start point to the goal state through the selected sequence of cells. This sequence ends in the current evaluated cell. The value $f(v)$ is the evaluation value of each adjacent cell of the current reached cell. The cell with the lowest value of $f(v)$ is chosen as the next one in the sequence (Duchoň et al., 2014).

The heuristic function $h(v)$ tells the A-star algorithm an estimate of the minimum distance from any vertex v to the goal. The chosen heuristic function controls A-star's behaviour. At one extreme, if $h(v)$ is zero, then only $g(v)$ plays a role, and the algorithm turns into Dijkstra's Algorithm, which is guaranteed to find a shortest path. If the $h(v)$ value always is lower or equal to the distance of moving from v to the goal, A-star is guaranteed to find a shortest path. The lower value of $h(v)$, the more vertices A-star expands, making the algorithm perform slower. If $h(v)$ is exactly equal to the distance of moving from v to the goal, then A-star will only follow the best path and never expand anything else, making the runtime of the algorithm short. Although this is not the case for most problems, it is possible to adjust the function to exact for special cases. If $h(v)$ is sometimes greater than the distance of moving from v to the goal, A-star is not guaranteed to find a shortest path, but the runtime of the algorithm is faster. At the other extreme, if $h(v)$ is very high relative to $g(v)$, then only $h(v)$ plays a role, and A-star turns into Greedy Best-First-Search (Patel, A., 2016).

The possibility of using different criterion of distances is an advantage of the A-star algorithm, which gives to it a wide range of modifications for this basic principle. For example, time, energy consumption or safety can be also included in function $f(v)$ (Duchoň et al., 2014).

4.6.3 Jump Point Search (JPS)

In this paper, we deal with such path symmetries by developing a “macro-operator” that selectively expands only certain vertices from the grid, which we call *jump points*. Moving from one jump point to the next involves travelling in a fixed direction while repeatedly applying a set of simple neighbour pruning rules until either a dead-end or a jump point is reached. Because we do not expand any intermediate vertices between jump points this strategy can have a dramatic positive effect on search performance. Furthermore, computed solutions are guaranteed to be optimal. Jump point pruning is fast, requires no pre-processing and introduces no memory overheads. It is also largely orthogonal to many existing speedup techniques applicable to grid maps.

Harabor & Grastien (2011) proposes a Jump point search strategy that speeds up optimal search by selectively expanding only certain vertices on a grid map, which we term jump points, when working with undirected uniform-cost grid maps. Each vertex has ≤ 8 neighbours and is either traversable or not. Moves involving non-traversable (obstacle) vertices are disallowed. The notation $\vec{d}$ refers to one of the eight allowable movement directions (up, down, left, right etc.). We write $y = x + k\vec{d}$ when vertex y can be reached by taking k unit moves from vertex x in direction $\vec{d}$. When $\vec{d}$ is a diagonal move, we denote the two straight moves at 45 degrees to $\vec{d}$ as $\overrightarrow{d_1}$ and $\overrightarrow{d_2}$. A path $p = (v_0, v_1, \ldots, v_k)$ is a cycle-free ordered walk starting at vertex v_0 and ending at v_k. The setminus operator is sometimes used in the context of a path: for example, $\pi \setminus x$. This means that the subtracted vertex x does not appear on the path. The function *len* is used to refer to the length of a path and the function *dist* to refer to the distance between two vertices on the grid: e.g., $len(\pi)$ or $dist(v_0, v_k)$ respectively. An example of the basic idea is shown in Figure 4.12 below, where obstacles or dead ends are coloured black.

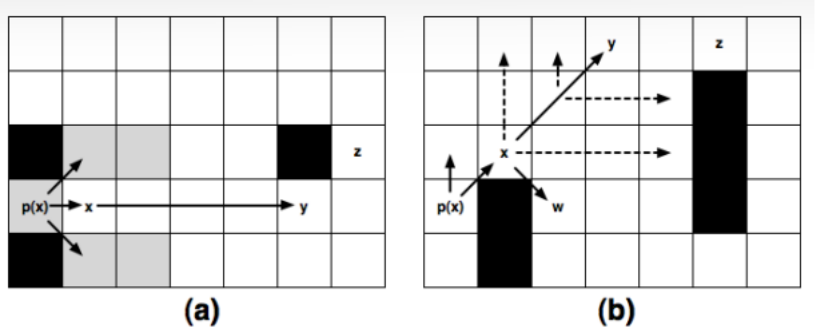


Figure 4.12 Examples of straight (a) and diagonal (b) jump points. Dashed lines indicate a sequence of interim vertex evaluations that reached a dead end. Strong lines indicate eventual successor vertices (source: Harabor & Grastien).

Here the search is expanding a vertex x which has $p(x)$ as its parent. The direction of travel from $p(x)$ to x is a straight move to the right. When expanding x it is noticeable that there is little point to evaluating any neighbour highlighted grey as the path induced by such a move is always dominated by (i.e., no better than) an alternative path which mentions $p(x)$ but not x, the only non-dominated neighbour of x lies immediately to the right. Rather than generating this neighbour and adding it to the open list, as in the classical A-star algorithm, jump point search rather step to the right and continue moving in this direction until we encounter a vertex such as y, which has at least one other non-dominated neighbour (here z). If we find a vertex such as y (a jump point) we generate it as a successor of x and assign it a g-value of $g(y) = g(x) + dist(x, y)$. Alternatively, if we reach an obstacle, we conclude that further search in this direction is fruitless and generate nothing (Harabor & Grastien, 2011).

Neighbour Pruning Rules

Rules for pruning are applied to the set of vertices immediately adjacent to some vertex x from the grid. The objective is to identify from each set of such neighbours, i.e., $neighbours\ (x)$, any vertex v that do not need to be evaluated in order to reach the goal optimally. We achieve this by comparing the length of two paths: π, which begins with vertex $p(x)$ visits x and ends with v and another path π' which also begins at vertex $p(x)$ and ends with v but does not mention x. Additionally, each vertex mentioned by either π or π' must belong to $neighbours\ (x)$ (Harabor & Grastien, 2011).

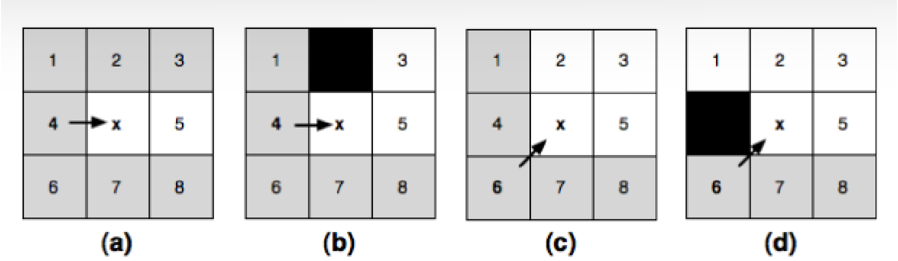


Figure 4.13 Several cases where a node *x* is reached from its parent by either a straight or diagonal move. When *x* is expanded all nodes marked grey can be pruned from consideration (source: Harabor & Grastien).

There are two cases to consider, depending on whether the transition to x from its parent $p(x)$ involves a straight move or a diagonal move. Note that if x is the start vertex, $p(x)$ is null and nothing is pruned.

Straight Moves: We prune any vertex $v \in neighbours(x)$ which satisfies the following dominance constraint:

$$len(\ \langle p(x), \dots, v\rangle \setminus x\ ) \leq len(\ \langle p(x), x, v\rangle\ ) \tag{13}$$

From Figure 5.13(a), we see that when $p(x) = 4$, we prune all neighbours except $v = 5$.

Diagonal Moves: This case is similar to the pruning rules developed for straight moves; the only difference is that the path which excludes x must be strictly dominant:

$$len(\ \langle p(x), \dots, v\rangle \setminus x\ ) < len(\ \langle p(x), x, v\rangle\ ) \tag{14}$$

Figure 5.13(c) shows an example. Here $p(x) = 6$ and we prune all neighbours except $v = 2$, $v = 3$ and $v = 5$ (Harabor & Grastien, 2011).

Assuming $neighbours(x)$ contains no obstacles, we will refer to the vertices that remain after the application of straight or diagonal pruning (as appropriate) as the *natural neighbours* of x. These correspond to the non-gray vertices in Figures 5.13(a) and 5.13(c). When $neighbours(x)$ contains an obstacle, we may not be able to prune all non-natural neighbours. If this occurs, we say that the evaluation of each such neighbour is *forced*. A vertex $v\ \in neighbours(x)$ is forced if v is not a natural neighbour of x:

$$len(\ \langle p(x), x, v\rangle\ ) < len(\ \langle p(x), \dots, v\rangle \setminus x\ ) \tag{15}$$

Figure 5.10(b) above shows an example of a straight move where the evaluation of $v = 3$ is forced. Figure 5.10(d) shows a similar example involving a diagonal move, here the evaluation of $v = 1$ is forced (Harabor & Grastien, 2011).

4.7 Multi-objective optimization methods

In a multi-objective optimization problem, we attempt to find the "best" solution(s) to a problem by maximizing or minimizing a set of objectives. Fundamentally, multi-objective optimization problems have two major components: design variables, and objectives. Design variables are parameters that the designer might "adjust" in order to modify the artifact he is designing, whereas the objectives are the mathematical representation of the designer's optimal solution. Multi-objective optimization algorithms allow designers to examine competing objectives and determine what compromises are necessary to create a valid design. This allows the designer to evaluate a set of optimal designs and determine which design aligns with their preferences and goals (Stewart et al., 2021).

4.7.1 Definition of a multi-objective optimization problem

The general multi-objective optimization problem is posed as follows:

$$Minimize\ \boldsymbol{F}(x) = \ [f_1(x), f_2(x), \dots, f_k(x)]^T \tag{16}$$

$$s.t.\ x \in S$$

$$x = (x_1, x_2, \dots, x_n)^T \tag{17}$$

Where $f_1(x), f_2(x), \dots, f_k(x)^T$ are the k objective functions, $(x_1, x_2, \dots, x_n)$ are the n optimization parameters, and $S \in R^n$ is the solution or parameter space. Obtainable objective vectors, $\{\mathbf{F(x)}x \in S\}$ are denoted by Y, so $\mathbf{F}: S\ \rightarrow Y, S$ is mapped by $\mathbf{F}$ onto Y
$Y \in R^k$ is usually referred to as the attribute space, where ∂Y is the boundary of Y. For a general design problem, $\mathbf{F}$ is non-linear and multi-modal, and S might be defined by nonlinear constraints containing both continuous and discrete member variables.
$f_1^*, f_2^*, \dots, f_k^*$ is used to denote the individual minima of each respective objective function, and the utopian solution is defined as $\mathbf{F}^* = (f_1^*, f_2^*, \dots, f_k^*)^T$. As $\mathbf{F}^*$simultaneously minimizes all objectives, it is an ideal solution that is rarely feasible. Figure 4.14 on the next page provides a visualization of the nomenclature.

In this formulation, minimize $\mathbf{F(x)}$, lacks clear meaning as the set $\{\mathbf{F(x)}\}$ for all feasible $\mathbf{x}$ lacks a natural ordering, whenever $\mathbf{F(x)}$ is vector valued. To determine whether $\mathbf{F(x_1)}$ is better then $\mathbf{F}(\mathbf{x_2})$, and thereby order the set $\{\mathbf{F}(\mathbf{x})\}$**,** the subjective judgment from a decision-maker is needed (Andersson, 2000).

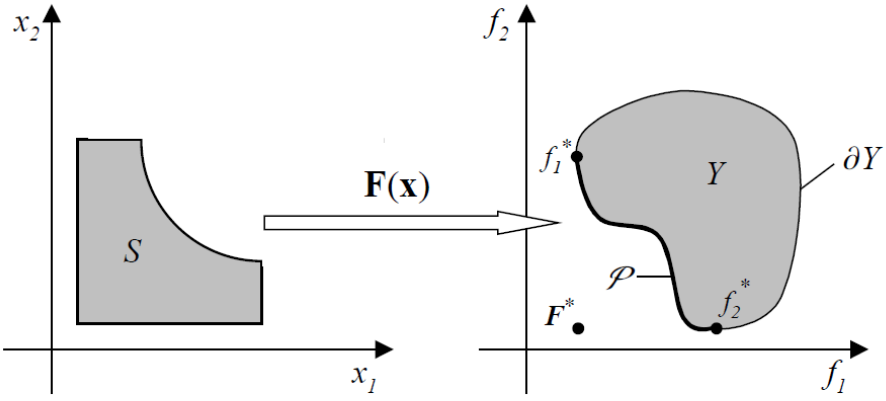


Figure 4.14 Solution and attribute space nomenclature for a two-dimensional problem with two objectives (source: Andersson, 2000).

4.7.2 Pareto Optimality

One property commonly considered as necessary for any candidate solution to the multi-objective problem is that the solution is not dominated. The Pareto set therefore consists of solutions that are not dominated by any other solutions. A solution **x** is said to dominate **y** if **x** is better or equal to **y** in all attributes, and strictly better in at least one attribute. In a minimization problem with two solution vectors **x, y** $\in S$. x is said to dominate **y**, denoted $\mathbf{x} \succ \mathbf{y}$, if:

$$\forall i \in \{1,2,\dots,k\}\colon f_i(\mathbf{x}) \leq f_i(\mathbf{y}) \quad and \quad \exists j \in \{1,2,\dots,k\}\colon f_j(\mathbf{x}) < f_j(\mathbf{y}) \tag{18}$$

The space in R^k is formed by the objective vectors of Pareto optimal solutions, this is known as the Pareto optimal frontier, $\mathcal{P}$. Pareto optimal solutions are also known as non-dominated or efficient solutions. In a final design solution, it is clear that the solution should preferably be a member of the Pareto optimal set, there would not exist any solutions that are better in all attributes. If the solution is not in the Pareto optimal set, it could be improved without degeneration in any of the objectives, and is not a rational choice (Andersson, 2000).

4.7.2.1 Testing for Pareto Optimality

From the literature we can find purposed methods for determining whether a point is Pareto optimal or not with the following test for the point $\mathbf{x}^*$:

$$\underset{\mathrm{x} \in \mathrm{X}, \boldsymbol{\delta} \geq 0}{\text{Minimize}} \sum_{i=1}^{k} \delta_i \tag{19}$$

Subject to

$$F_i(\mathrm{x}) + \delta_i = F_i(\mathrm{x}^*), \quad i = 1,2, \dots, k.$$

If all δ_i are zero, then x^* is a Pareto optimal point. For any given problem, there may be an infinite number of Pareto optimal points constituting the Pareto optimal set. Therefore, one must distinguish between methods that provide the Pareto optimal set or some portion of that set, and methods that seek a single final point (Marler & Arora, 2004).

4.7.3 Different ways to perform multi-objective optimization

As most optimization problems are multi objective to their nature, there are many methods available to tackle these kinds of problems. The multi-objective optimization problem can be handled in four different ways depending on when the decision-maker articulates his or her preference on the different objectives. This can be never, before, during or after the actual optimization procedure. *No preference articulation* methods do not use any preference information from the decision maker before conducting the optimization procedure. *Priori* articulation of preference information is the most common way of conducting multi-objective optimization. This means that before the actual optimization is conducted the different objectives are somehow aggregated to one single figure of merit. *Progressive* articulation of preference information methods is generally referred to as interactive methods. They rely on progressive information about the decision-makers preferences simultaneously as they go through the optimization procedure. *Posteriori* articulation of preference information gives the decision-maker the possibility of choosing between different Pareto optimal solutions that are presented to him or her. In this type of methods, the solution is independent of the decision-makers preferences (Andersson, 2000). The focus of this thesis is methods with priori articulation of preference information. Most of these methods incorporate parameters, which are coefficients, exponents, constraint limit, etc. that can either be set to reflect the decision-makers preferences or be continuously altered to represent the complete Pareto optimal set (Marler & Arora, 2004). Further, three priori articulation methods considered for the thesis is described, the *weighted global criterion method*, the *weighted sum approach* and *goal programming*.

4.7.3.1 Weighted global criterion method

One of them most common general scalarization methods for multi-objective optimization is the *global criterion method* in which all objective functions are combined to from a single function. Although a global criterion may be a mathematical function with no correlation to preferences, a weighted global criterion is a type of utility function in which method parameters are used to model preferences. One of the most general utility functions is expressed in its simplest form as the *weighted exponential sum* (Marler & Arora, 2004):

$$U = \sum_{i=1}^{k} w_i [F_i(\mathrm{x})]^p \,, \quad F_i(\mathrm{x}) > 0 \forall i, \tag{20}$$

$$U = \sum_{i=1}^{k} [w_i F_i(\mathrm{x})]^p \,, \quad F_i(\mathrm{x}) > 0 \forall i, \tag{21}$$

The most common extensions of (20) and (21) purposed in the literature are:

$$U = \left\{ \sum_{i=1}^{k} w_i [F_i(\mathrm{x}) - F_i^{\circ}]^p \right\}^{\frac{1}{p}} \,, \tag{22}$$

$$U = \left\{ \sum_{i=1}^{k} w_i^p [F_i(\mathrm{x}) - F_i^{\circ}]^p \right\}^{\frac{1}{p}} \,. \tag{23}$$

Here, w is a vector of weights typically set by the decision maker such that $\sum_{i=1}^{k} w_i = 1$ and $\mathrm{w} > 0$. Generally, the relative value of the weights reflects the relative importance of the objectives. One can view the summation arguments, which are shown in equations (22) and (23), in two ways: as transformations of the original objective functions, or as components of a distance function that minimizes the distance between the solution point and the utopia point. As the decision-maker usually has to compromise between the final solution and the utopia point, global criterion methods are often called utopia point methods or compromise programming methods (Marler & Arora, 2004).

The solution to these approaches depends on the value of p. Generally, p is proportional to the amount of emphasis placed on minimizing the function with the largest difference between $F_i(\mathrm{x})$ and F_i°. Global criteria yield portions of the Pareto optimal set with continuous variation in p. However, varying only p usually yields only a limited number of Pareto optimal solution points. When using these methods, p and $\mathbf{w}$ are typically not varied or determined in unison as one might think. Rather, one usually selects a fixed value for p. Then, the user either sets $\mathbf{w}$ to reflect preferences a priori or systematically alters $\mathbf{w}$ to yield a set of Pareto points. For all these methods, using higher values for p increases the effectiveness of the method in providing the complete Pareto set. Equation (22) is sufficient for Pareto optimality as long as $\mathbf{w} > 0$. (23) is also sufficient for Pareto optimality. (21) which is similar to (23), provides a necessary condition for Pareto optimality assuming $\mathbf{F}(\mathrm{x}) \geq \mathbf{0}$. Technically, this means that for each Pareto optimal point x_p, there exists a vector $\mathbf{w}$ and a scalar p such that x_p is the solution to (21). Generally, using a higher value of p enables one to better capture all Pareto optimal points, doing so may also yield non-Pareto optimal points (Marler & Arora, 2004).

4.7.3.2 Weighted-sum approach

The weighted-sum method for priori articulation optimization utilizes linear combination of weights and optimization functions to find optimal solutions. The objective function is formulated as a weighted $\mathcal{L}_1$-metric, given by the Equation (24) below:

$$min \sum_{i=1}^{k} w_i f_i(\mathbf{x}) \tag{24}$$

Subject to

$$\mathbf{x} \in S$$

$$w \in R^k | w_i > 0, \sum w_i = 1$$

Where w_i is the weight for the ith objective function, $f_j(x)$, and $w_i > 0$ for all i. The larger the w_i, the more important f_i is to the problem, which means f_i will have a larger effect on the set of solutions that are obtained. The fact that $w_i > 0$ substantiates the fact that the objective function f_i plays a role in determining an optimal solution (Stewart et al., 2021).

By choosing different weights, w_i, for the different objectives, the preferences of the decision-maker are considered. As the objective functions are generally of different magnitudes, they might have to be normalized first (Andersson, 2000).

This method removes much of the complexity in multi-objective optimization and is fairly easy to implement for most problems. One setback of this method is that the burden of assigning the weights fall on the decision-maker. Assigning inappropriate weights can lead to a Pareto front that fails to meet the decision-makers goals and may require the decision-maker to iterate the weights (Stewart et al., 2021).

4.7.3.3 Goal programming

In goal programming the objectives are formulated as goal criteria that the decision maker wants each objective to possess. The criteria could be formulated in the following ways. We want the objective to be:

1. Greater than or equal to
2. Less than or equal to
3. Equal to
4. In the range of

Usually, a point that satisfies all goals is not a feasible solution. The goal programming problem is to find the point in S whose criterion vector "best" compared with the utopian set, i.e., has the smallest deviation from the utopian solution. There are different ways to determine which point in S that compares best with utopian solution based on different goal programming methods. One of these goal programming methods is the Archimedean, which uses a weighted $\mathcal{L}_1$-metric to determine the "best" solution. In this case the objective function is reformulated as a weighted sum of undesired deviations from the utopian solution set. Another goal programming method is the lexicographic, where the goals are grouped according to priorities. In the first stage a set of solutions that minimize the deviation to the goal with the highest priority is obtained. In the second stage this set of solution is searched through to find a subset that minimizes the deviation from the second most important goal. The process continues until only one single point is left as the final solution (Andersson, 2000).

The idea to minimize the deviation from a utopian solution set is what makes goal programming very attractive. Some of the problems with goal programming is that if the initial set of weights in the Archimedean approach does not obtain a satisfying solution, the weights must be changed in order to obtain a better solution without any clear direction in which the weights should be changed. A problem with lexicographic goal programming is that lower priority goals might not influence the generated solution, as a single point might be found without having to consider all goal criteria (Andersson, 2000).

4.8 Solution Methods Used

Pei et al. (2020) presents a Minimizing Collective dose Path-Planning method (MCPP) to reasonably arrange personnel evacuations and reduce the exposure of personnel during the process. In this method, the traditional Dijkstra algorithm is used to find the shortest path between nodes in a graph. After establishing the evacuation network, the accumulated dose per section were first calculated, and the accumulated dose-based Dijkstra algorithm was proposed to find N optimal routes in turn. Then the equilibrium model considering road resistance was used to make reasonable arrangements so that all people can complete the evacuation in the limited time with the minimum collective dose. Finally, an optimized evacuation plan was obtained. However, the MCPP method proposed is meant to provide support for decision making in the early stage of an emergency response on nuclear plants. In this thesis we are interested in finding efficient paths, which can be provided with short computational times, that can be used as decision support when deciding on which path to use for a desired destination or a roundtrip where several points need to be visited. The advantage of the model provided, is that it shows that Dijkstra algorithm is a good starting point for further testing, since it gives good results for path planning problems that includes radiation.

Sapundzhi & Popstoilov (2017) presents four different optimization algorithms for finding the shortest paths. The algorithms of Dijkstra, Floyd-Warshall, Bellman-Ford and Dantzig were examined and analysed. The best results for the values of execution time and different number of vertices are obtained by Dijkstra’s algorithm.

This algorithm was also implemented through an adjacency matrix, where better results was obtained when using the adjacency matrix for the same input parameters. It was here shown that the time complexity of the Dijkstra's algorithm depends on the number of vertices and is inversely proportional to the number of vertices. The execution time with fifty vertices in seconds were 0.173, compared to 10 000 vertices the execution time was 81.373. Even though this study only considers one objective, it is suitable for further testing of Dijkstra algorithm.

Chen et al. (2020) proposes an improved A-star algorithm for searching the minimum dose path in nuclear facilities. Unlike the traditional A-star algorithm, which is vulnerable to the interference of the complex environment, the improved one has a stronger anti-interference ability. The improved algorithm presented can obtain the least cumulative dose and the path length is appropriate, which can adapt to the complex environment in nuclear power plants. However, the results presented with this algorithm does not have as low computational times as desired for our problem, but it shows that a modified version of the A-star algorithm is suitable for further development in dose path planning.

Duchoň et al. (2014) introduces several modifications and improvements of A-star algorithm when dealing with path planning of a mobile robot based on a grid map. The performance of the algorithms was evaluated by computational time, length of path, number of examined cells and symmetry of the examined environment. Individual modifications were evaluated in several scenarios, which varied in the complexity of the environment. Based on these evaluations, it is purposed that it's possible to choose a path planning method suitable for individual scenarios. In terms of finding the path fast, Jump Point Search (JPS) was the best algorithm in various environments. This algorithm is therefore suitable to use in cases where it is necessary to quickly find a path, where computational times are significant. The Jump Point Search method is suitable for further development since it's fast and works well with path planning on a grid map.

All the algorithms have their own advantages and drawbacks. Some of them work well for narrow spaces and are suitable for corridors, but at the same time can barely find a path in the open space. Because of this, we desire to develop a tool suitable for testing different algorithms on individual scenarios, where it's possible to make the path decision based on own criterion.

Constructing a Pareto front would help visualize the solutions and make it easier for the decision maker to choose between the available solutions. After reviewing the different multi-objective optimization methods. The weighted sum approach was chosen as the most fitting since this gives the decision maker freedom to assign suited weights to the objective function according to own preferences of which of the goals that are most important. Marler & Arora (2009) presents the weighted sum method for multi-objective optimization stating that if all weights are positive, then minimizing the composite objective function as shown in previous chapter provides a sufficient condition for Pareto optimality.

5. Methodology

In the previous chapters, we presented a problem of the shortest path in the radiation environment. In this chapter, we describe the software developed to solve the problem. The absence of any existing available software solution has set a non-trivial task to develop from scratch such an application. In this work, we developed a minimum valuable product that provides tools for creating elementary graph maps, implementation of several pathfinding algorithms in a radiation environment, and panels to visualize the paths and compare the results. This software allows the user to make various experiments with already known algorithms and a modified version of the Jump Point Search algorithm, which is our major contribution from the algorithmic perspective. An overview is provided in Sections 5.1 and 5.2.

5.1 Program Description

The implementation of the algorithms for the proposed problem requires software based on two main pillars: a platform for graph manipulation and software for radiation simulation. At the current moment, there is no available software that consists of both necessary parts. Independent development of a tool for radiation simulation is a challenging task that requires a considerably large amount of time and is out of our scope. Therefore, our own solution was developed in a collaboration with the Institute for Energy Technology (IFE), who kindly shared with us the radiation simulation part of their commercial product the VRdose. Since this platform is written in Java, the whole application was also implemented in Java. An inseparable part of a program for such goal is visualization, which makes the pathfinding process more comprehensible, especially when there is nuclear pollution in an instance. There is a wide range of user interface (UI) packages for Java, but the most contemporary is Java FX. Java FX is a software for creating desktop and mobile applications with modern design and impressive performance, thus we have chosen this platform for the application implementation.

5.2 RadPathFinder

The developed RadPathFinder application consists of two main sections: The Graph editor and Launcher. In the next sections, we present the functional features of the application.

5.2.1 Graph editor

The graph editor has several tools to manipulate a grid graph, such as shields, radiation sources, and start and endpoint allocations. Next, we are going to describe the workflow to use the software where a basic overview is shown in Figure 5.1 below.

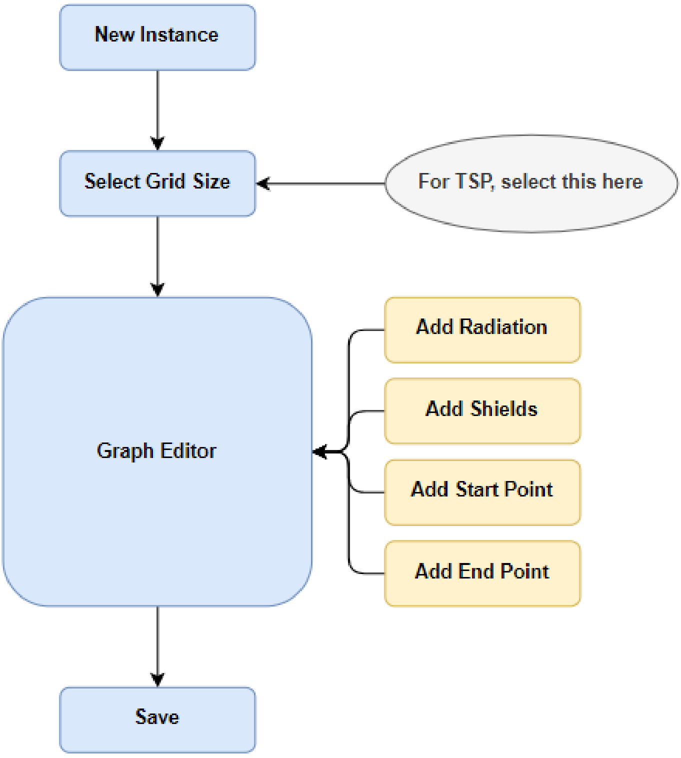


Figure 5.1 Flowchart of how to create a new instance in the graph editor.

In the beginning, the user needs to choose the size of the grid and enable the TSP mode. The TSP mode implies the allocation of several endpoints without a limit on the number. The size of the grid map is discrete varying by one meter, and the recommended upper limit for the grid size is 100 × 100 meters. The meters are realistic, where each meter is represented with cell size of 0.1 × 0.1 meters. Thus, the grid of size 100 × 100 meters contain 1 million vertices. When this is done, a new window of the editor appears.

The user can observe a viewport with the whole grid and a tool panel with the available modifications of the grid, such as radiation sources and shields. The application has 5 different types of shields: water, sand, concrete, iron, and lead. The radiation sources are presented with 118 various isotopes, for instance, C-15, Na-22, Co-58, etc.

To allocate a shield or a radiation source, the user needs to select the desired element, specify the intensity of radiation in cases with radiation sources, and stretch the object to the desired size on the grid. The size and number of objects are limited only by grid size. At any moment of the grid design, the user can compute radiation concentration for all vertices on the grid and then select a heatmap representation if this is desired. The heatmap changes its colour based on the radiation level at each point of the map. In case the results are unsatisfactory, the user can modify the grid until the desired result is achieved. The start and endpoints are allocated in the same way as obstacles but just as points on the grid.

5.2.2 Launcher

The launcher is divided into main two parts: 1. A viewport for the grid and 2. A view list with all the paths created during a session. The user can choose between all available algorithms and launch them one by one to compare the results. For the TSP problem, there are four different view lists, the first displays all computed paths between each vertex, the second shows only paths included in a Hamiltonian path optimized according to distance, the third does the same as the second but for a path optimized according to accumulated radiation, and the last displays a path for multi-objective optimization of both radiation and distance.

During the search, the viewport displays all the manipulations executed on the grid, which contributes to understanding the search process and bug fixing. Moreover, when a path is found, it is displayed on the viewport. Afterward, all the results are saved into an excel file. The excel file contains information: the name of instance, type of algorithm, heuristic used, size of the grid, elapsed time, path length, radiation and distance weights, and accumulated radiation. A basic overview of the launcher is shown in Figure 5.2 below.

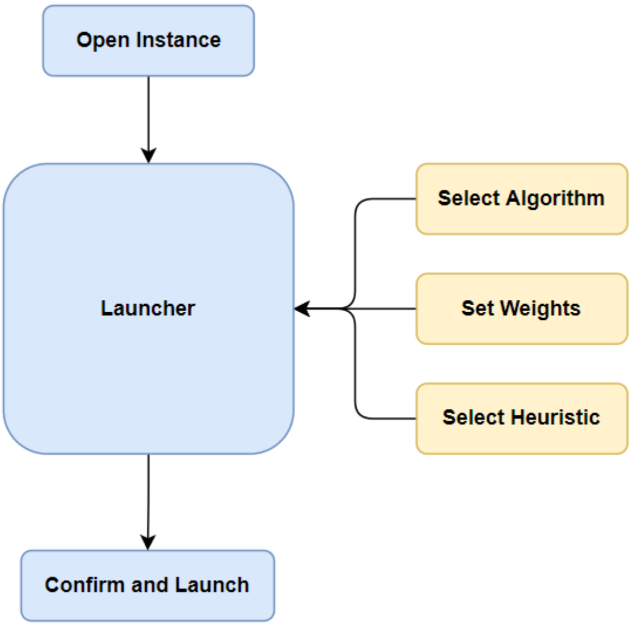


Figure 5.2 Flowchart of how to use the Launcher.

6. Implementation Details

This chapter describes the implementation of the grid map, described in Chapter 4, and the modification of the Dijkstra, A-star, and Jump Point Search algorithms for pathfinding in a radiation environment. It also gives a short overview of the radiation simulation package VRdose. The software described in Chapter 5 includes our implementation of six algorithms for pathfinding, where three of them are adapted for radiation environments. The next section gives a brief overview of the key structures of the application. All of them are essential and form the base for the further implementation of any pathfinding algorithms. They are implemented with the usage of interfaces that provides good scaling opportunities for further research.

Section 6.1 presents the key structures in the model. Section 6.2 presents the radiation simulation package used in the model. The different algorithms used in the model and modifications of these are presented in Section 6.3. Section 6.4 describes the traveling salesman problem for the model.

6.1 Key Structures

Vertex

Vertex or node is the fundamental unit of a graph. In our implementation, each vertex contains information about its position on the grid, the type of its material, its parent, and the radiation level in its position. The vertexes are created when an instance of grid is initialized.

Material

Material is a presentation of a vertex structure. There are several types of materials: radiation source, water, iron, concrete, lead, and sand. The definition of the material is necessary for vertexes to provide correct presentation of the vertexes in a viewport.

Grid map

The grid map is presented as a two-dimensional matrix, and each cell consists of a vertex. A vertex has between three and eight neighbors. In addition, the grid has information about all its radiation sources and protection shields. The references on the start and end points are stored as well. The grid performs most manipulations with vertexes, such as providing neighbouring vertexes, changing materials during the creation and modification of a grid map, providing information about the level of radiation in a particular place on the map. By default, each cell describes 0.1 meters of space.

Radiation source

Radiation source is an object which has information about its size and position in the space, isotope type, and intensity of radiation, which is necessary to create the radiation scene. The current implementation consists of 118 various isotopes.

Shield

There are five different types of shielding materials, which are mentioned in the materials section. Each instance of shield has information about its size, position, and thickness, which defines its protection capacity.

Radiation scene

The radiation scene is an object that combines all traversable vertices, radiation sources, and protecting shields. The radiation scene is a connecting link between the VRdose package and our application. It collects information about all points and objects on the grid and transforms them into a suitable format for the VRdose. The results of the radiation simulation are received and handled by this class.

Path

A path contains all vertices that are assigned to it, computes, and stores its distance and accumulated radiation.

6.2 VRdose package

The VRdose package uses a new Point Kernel Approach for the radiation simulation. The point kernel approach quantifies the radiation burden originating from gamma sources based on the distance between the source and the detector, taking into account attenuation and build-up of photons in the objects (shields) intersecting the direct path of radiation to the detector. Due to a simple theoretical approach, point kernel techniques are fast. However, because of the simplifications introduced, they are less precise and may break down in specific situations, e.g., very short distances between source and detector (Szöke et al., 2014). The theoretical assumptions underlying this dosimetry model are described below. The model is based on a more up-to-date and flexible application of the basic point kernel approximation (25):

$$R = \sum\nolimits_i \left(\frac{Ay_i}{4\pi r^2} \times \exp\left(-\sum r(E_i)\right) \times CF(E_i) \times B\left(\sum r(E_i)\right)\right) \tag{25}$$

where,

- $R -$ is the detector (point isotropic) response (dose),
- $S -$ is a set of radiation sources on a map,
- $E_i -$ is the photon energy, $i \in S$,
- $A -$ is the activity of the source (point isotropic) (Bq),
- $y_i -$ is the yield of photons at photon energy E_i, $i \in S$,
- $r -$ is the distance from the source to the detector (cm),
- $\sum r -$ is the optical thickness of the medium between the detector and the source (including air),
- $CF -$ is the conversion coefficient at energy E_i, $i \in S$,
- $B(\sum r(E_i)) -$ is a suitable buildup factor, $i \in S$ (Szöke et al., 2014).

6.3 Modified algorithms for the RadPathFinder

Chapter 4 gave an overview of the main pathfinding algorithms. All these algorithms find optimal distances under certain conditions. Unfortunately, an optimal path in terms of distance is not guaranteed to be optimal in terms of accumulated radiation. Even though accumulated radiation increases with the growth of distance, in many cases a longer path is safer than the shortest. In this section, we describe modifications of the previously stated algorithms in order to work in a radiation environment with optimization according to accumulated radiation or radiation and distance.

6.3.1 Dijkstra algorithm

The basic Dijkstra algorithm is described in detail in Chapter 4. The Dijkstra algorithm finds an optimal path according to the distance by sequential exploration of the vertices on a graph. Consequently, it is possible to modify the algorithm and make it work with accumulated radiation instead of distance. There are different versions of the algorithm, but the most common is to find the shortest path from the source vertex to all other vertices in the graph. The time complexity also varies depending on the implementation. The base average time complexity is $O(V^2)$, but it is possible to reduce it to $O(V + E\log V)$, where V – is a number of vertices, and E – is a number of edges. This improvement can be achieved using the usage of a min-priority queue. Pseudocode 1 presents the main loop of the algorithm, and Pseudocode 2 shows how vertices are explored.

Main definitions of the Dijkstra algorithm with accumulated radiation:

- $G = (V, E)$, where G is a graph formed by vertices V and edges E,
- $source$ – is the start vertex,
- $current$ – is the vertex currently expanding,
- rad_i – is a HashMap to store accumulated radiation of vertices i, $i \in V$,
- *open* – is a min-priority queue of reached but not expanded vertices, first vertex in the queue is a vertex with minimum value of accumulated radiation,
- *closed* – is a set of reached vertices,
- *endVertex* – is an end vertex, the vertex an algorithm looks for.

Pseudocode 1: Dijkstra algorithm with accumulated radiation, main loop

```
Input: G = (V, E)
*Initialization of open, rad, closed;
$rad_{source}$ =0;
$rad_{v}$ =+infinity for all $v \in V/\{source\}$ //*Set accumulated radiation for vertices
open.add(source);
while(!open.isEmpty() && !closed.contains(endVertex)){
    current = open.getFirst();
    closed.add(current)
    updateRadiation();
    open.remove(current);
}
radiation.get(endVertex); // get accumulated radiation at the end vertex
```

Pseudocode 2: Dijkstra algorithm with accumulated radiation, updateRadiation();

```
For each $neighbor \in current$ {
If (!closed.contains(neighbor) && $rad_{current} + getRadiation(current, neighbor) < rad_{neighbor}${
     $rad_{neighbor}$.replace($rad_{current}$ + getRadiation(current, neighbor));
     neighbor.setParent(current);
     open.add(neighbor);}
}
```

The main difference from a classic Dijkstra algorithm is the way how the accumulated radiation is computed between vertices. For testing purposes, there is no reason to complicate the process computation of accumulated radiation. The high precision in calculations of accumulated radiation is essential in software packages that simulate the effect of radiation on humans or robots. In such a case, an author needs to only rewrite the method getRadiation() based on their necessity in precision. Pseudocode 3 demonstrates our implementation of the getRadiation function. There is no Sum-Weighted method for the Dijkstra algorithm as the A-star does the same work and gives the same results.

Pseudocode 3: Dijkstra algorithm with accumulated radiation, getRadiation();

```
getRadition(current, neighbor){
    Switch(distance type)
        case Euclidian:
            time = getEuclidianDistance(current, neighbor) / speed;
            return time * neighbor.getRadiationLevel();
        case Manhattan:
            time = getManhattanDistance(current, neighbor) / speed;
            return time * neighbor.getRadiationLevel();
}
```

6.3.2 A-star algorithm

The A-star algorithm is described in detail in Chapter 4. In this section, we will focus on the modifications of the algorithm that make it possible to find an optimal path in a radiation environment.

The main difference between the A-star from Dijkstra is a heuristic function that leads the search. The most common are Euclidean and Manhattan distances that are computed between a currently explored vertex and the end vertex. This distance is afterward assigned to the current vertex. The usage of such heuristic functions significantly speeds up the search, which makes the implementation of A-star quite promising. Unfortunately, there is no heuristic described in the literature that can lead search based on radiation contamination of the environment. Subsequently, there is no way to increase the speed performance of the algorithm. Sequential exploration of vertices leads to the same performance as the Dijkstra algorithm.

The absence of a heuristic for efficient exploration of space with radiation contamination does not mean that the implementation of A-star is unreasonable and fruitless, and all the necessary computations can be done with Dijkstra. On the contrary, since the A-star structure naturally fits into a realization of multi-objective function. The developed application has at its disposal both a classic A-star algorithm and its multi-objective realization. The classic version is well known and described in detail in Chapter 4, while the modified version is accompanied by some difficulties described further.

The multi-objectiveness is achieved with the use of the Weighted Sum method described in Chapter 4. In the implemented A-star algorithm, at each stage of the main loop, Equation (26) below computes the total weight v for each neighbor of the current vertex and adds them to an open list of vertices. At the beginning of each loop, iteration Equation (27) finds the next vertex in the open list to be explored with the minimum total weight v received from Equation 26.

$$v(i) = w_1 r_i + w_2 d_i + w_2 h D_i \tag{26}$$

where,

- i – a currently explored vertex.
- v – is the total weight of a vertex i,
- w_1 – is weight of radiation,
- w_2 – is weight of distance,
- r_i – is accumulated radiation at vertex i,
- d_i - is passed distance from the starting vertex to vertex i,
- hD_i – is a heuristic value of distance from vertex i to the end vertex.

$$n = argmin(v_i), \qquad i \in O \tag{27}$$

where,

- O – is a set of vertices in the open list,
- n – is a vertex with the minimal total value v,
- v_i – is the total weight of vertex i.

The weights for the method shall satisfy the following constraint:

$$w_1 + w_2 = 1,\ w_1, w_2 > 0;$$

Even though it may seem that the choice of weights is not a difficult task, in fact, it can be quite uneasy to solve. A fundamental deficiency in the weighted sum method is that it can be difficult to discern between setting weights to compensate for differences in objective-function magnitudes and setting weights to indicate the relative importance of an objective as is done with the rating methods. When objective functions have different ranges and orders of magnitude, an appropriate value for an objective function may not be apparent (Marler & Arora, 2009).

For example, consider our two objective functions to be minimized, where function-one is the distance, the range can be from 0.1 m. to 1 000 m., and function-two which represents accumulated radiation with the range from 0 to potentially infinity, but in most cases from 0 to 1 mSv. If accumulated radiation were decreased by half of a unit, it is a significant improvement. At the same time, a similar change in distance is insignificant. This leads to a conclusion that the results do not directly correspond to the importance of the goals.

The weights are severely affected by the absolute values of the objective functions. Particularly our problem is highly exposed to the drawback described above. The solution to this deficiency is to find the respective maximum for both accumulated radiation and distance and then divide the functions by the maxima. In such a case the objective functions are normalized and have ranged between 0 and 1 as suggested by Marler & Arora (2009).

Main definitions of the A-star algorithm with Sum-Weighted method

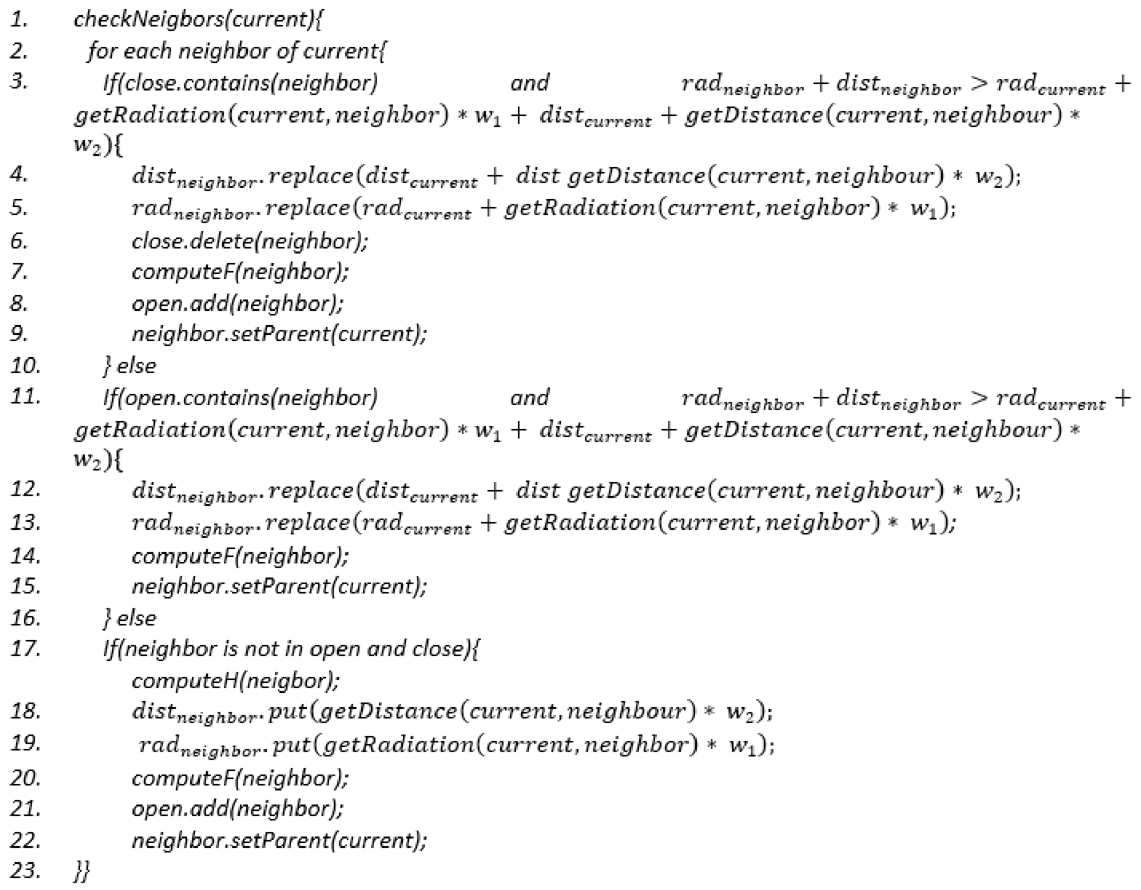
- $G \ = (V, E)$, where G is a graph formed by vertices V and edges E,
- $source$ – is a start vertex,
- $current$ – is a vertex currently expanding,
- rad_i – is a HashMap to store accumulated radiation of vertices i, $i \in V$,
- $dist_i$ – is a HashMap to store distances between source and vertex i, $i \in V$,
- $fList_i$ – is a HashMap to store total weight of vertex i, $i \in V$,
- $hDist_i$ – is a HashMap to store heuristic distance between vertex *i* and end vertex, $i \in V$,
- $open$ – is a min-priority queue of reached but not expanded vertices, first vertex in the queue is a vertex with minimum value of total weight in *fList*,
- $closed$ – is a set of reached and expanded vertices,
- $endVertex$ – is an end vertex, the vertex an algorithm looks for.

Pseudocode 4: A-star algorithm with Sum-Weighted method, main loop

```
Input: G = (V, E): the input graph
*Initialization of rad, dist, fList, hDist, open, closed;
$rad_{source} = 0$;
$dist_{source} = 0$
computeH(source);
computeF(source);
while(!open.isEmpty()) {
    current = open.pull();          // get a first vertex in the queue
    if (current != endVertex) {
        open.remove(current);        //remove current vertex from the queue
        checkNeigbors(current);      //expand current vertex
        close.add(current)           // add current vertex to the set of expanded vertices;
    } else {
        break;
    } }
```

Pseudocode 4 presents the main loop of the implemented A-star algorithm. There are three functions that require additional explanation; The function “computeH(current)” computes the heuristic distance between currently explored vertex and an endpoint. The function “computeF(current)” is the function 26 that is presented in this section above. The function “checkNeigbors (current)” presents the greatest interest as it defines the manipulation with vertices and the rules how they are updated, this function is presented in Pseudocode 5.

Pseudocode 5: A-star algorithm with Sum-Weighted method, checkNeigbors()



```
checkNeigbors(current){
  for each neighbor of current{
    If(close.contains(neighbor) and $rad_{neighbor} + dist_{neighbor} > rad_{current} + getRadiation(current, neighbor) * w_1 + dist_{current} + getDistance(current, neighbour) * w_2$){
        $dist_{neighbor}.replace(dist_{current} + dist\ getDistance(current, neighbour) * w_2)$;
        $rad_{neighbor}.replace(rad_{current} + getRadiation(current, neighbor) * w_1)$;
        close.delete(neighbor);
        computeF(neighbor);
        open.add(neighbor);
        neighbor.setParent(current);
    } else
    If(open.contains(neighbor) and $rad_{neighbor} + dist_{neighbor} > rad_{current} + getRadiation(current, neighbor) * w_1 + dist_{current} + getDistance(current, neighbour) * w_2$){
        $dist_{neighbor}.replace(dist_{current} + dist\ getDistance(current, neighbour) * w_2)$;
        $rad_{neighbor}.replace(rad_{current} + getRadiation(current, neighbor) * w_1)$;
        computeF(neighbor);
        neighbor.setParent(current);
    } else
    If(neighbor is not in open and close){
        computeH(neigbor);
        $dist_{neighbor}.put(getDistance(current, neighbour) * w_2)$;
        $rad_{neighbor}.put(getRadiation(current, neighbor) * w_1)$;
        computeF(neighbor);
        open.add(neighbor);
        neighbor.setParent(current);
}}
```

6.3.3 Jump Point Search

In the previous sections, two of the most known pathfinding algorithms and their adaptations for radiation environment have been described in detail. The Dijkstra algorithm finds optimal paths according to distance or accumulated radiation, which are good, but not always suitable. The optimal paths for accumulated radiation are not always reproducible by drones or human beings due to a high number of turns as they resemble zigzags or become too long. The A-star algorithm with the Sum-Weighted method solves this deficiency and finds the solutions belonging to the Pareto front and the user can find the most suitable weights for a particular problem. At the same time, the algorithm is not without flaws. The computation time is almost on the same level as Dijkstra, but to get benefits from the weights the user should preliminarily solve the Dijkstra algorithm with objectives function for distance and accumulated radiation. All these imperfections create a necessity for a faster algorithm with not optimal, but considerably good solution.

Jump point search is an algorithm that has not been widespread so far. The base implementation of the algorithm for distance is presented in Chapter 4. In this section, we present a modification of the Jump Point search algorithm for pathfinding in radiation environment. The algorithm recursively explores the grid map and looks for non-dominated neighbors which then are added to the open list, in this part the algorithm is identical to the classic implementation. The difference is that we do not rely on distance from the source to the currently expanded vertex and the heuristic distance between the current vertex and end vertex. Basically, we substitute the distances with accumulated radiation. For each non-dominated vertex of a current vertex, we compute accumulated radiation going directly from the current vertex to the non-dominated vertex. After, we add the non-dominated vertex to the open list and save their accumulated radiation from the source through all parents. The next step of the search starts with the choice of a non-dominated vertex with the lowest accumulated radiation among those in the open list. If during the search for a vertex in the open list a path with lower accumulated radiation is found, the radiation level and parent are updated. The procedure continues until the end vertex is found. The structure of the algorithm is similar to A-star, but the graph exploration procedure differs significantly. The presented Pseudocodes 6-9 shows the algorithm in more details.

Main definitions of the Jump Point Search method with accumulated radiation:

- $G = (V, E)$, where G is a graph formed by vertices V and edges E;
- $source$ – is a start vertex;
- $current$ – is a vertex currently expanding;
- rad_i – is a HashMap to store accumulated radiation of vertices i, $i \in V$;
- $dist_i$ – is a HashMap to store distances between source and vertex i, $i \in V$;
- $open$ – is a min-priority queue of reached but not expanded vertices, first vertex in the queue is a vertex with minimum value of total weight in fList;
- $closed$ – is a set of reached and expanded vertices;
- $neighbors_i$ – is a list of neighbors of current vertex, $i \in current$.
- $nonDVertex$ – is a non-dominated neighbor;
- *endVertex* – is an end vertex, the vertex an algorithm looks for.

Pseudocode 6: Jump Point Search with accumulated radiation, main loop

```
Input: G = (V, E): the input graph
*Initialization of rad, open, closed, goals;
goals = endVertex.getNeighbors();
open.add(source);
$rad_{source}$=0;
while(!open.isEmpty()) {
    current = open.get(); // get a first vertex in the queue
    close.add(current);
   If (current == endVertex){
      createPath();
      break;}
   findSuccessors();
}
```

Pseudocode 7: Jump Point Search with accumulated radiation, findSuccessors()

```
findSuccessors(){
neighbors = findNeighbors(current);
  for each $neighbor \in neighbours${
    nonDVertex = makeJump(neighbor, current, goals);
    If (nonDVertex != null) continue; //skip iteration
    accRad = creatPath(current, nonDVertex) + $rad_{current}$);
    If(closed.contains(nonDVertex) && accRad <$rad_{current}$){
      $rad_{nonDVertex}$ = accRad;
      nonDVertex.setParent(current);
      open.add(nonDVertex);
      close.remove(nonDVertex); }
    else if (closed.contains(nonDVertex)) {continue;}
    If(!open.contains(nonDVertex) || accRad < $rad_{current}$){
      $rad_{nonDVertex}$ = accRad;
      nonDVertex.setParent(current);
    If( closed.contains(nonDVertex ()){
      open.add(current);
      open.add(nonDVertex);
      close.remove(current);}
}}}
```

Pseudocode 8: Jump Point Search with accumulated radiation, makeJump()

```
makeJump(neighbor, current, goals){
If(neighbour != null || !walkable(neighbor)) return null;
If(goals.cointains(neighbor)) return neighbor;
  dx = neighbor.getX – current.getX;
  dy = neighbor.getY – current.getY;
  if (dx!=0 && dy!=0){
    If((walkable(neighbor.X – dx, neighbor.Y +dy)&&
    !walkable(neighbor.X -dx, neighbor.Y)) ||
    (walkable(neighbor.X + dx, neighbor.Y -dy)&&
    !walkable(neighbor.X, neighbor.Y -dy))) { return neighbor}
    If (makeJump(getVertex(neighbor.X + dx, neighbor.Y), neighbor, goals) != null ||
        makeJump(getVertex(neighbor.X, neighbor.Y + dy), neighbor, goals) !=null)
        { return neighbor};
  } else {
    If (dx != 0) {
    If((walkable(neighbor.X +dx , neighbor.Y + 1)&&
    !walkable(neighbor.X , neighbor.Y +1)) ||
    (walkable(neighbor.X + dx, neighbor.Y -1)&&
    !walkable(neighbor.X, neighbor.Y -1))) { return neighbor}
  } else {
    If((walkable(neighbor.X +1 , neighbor.Y + dy)&&
    !walkable(neighbor.X+1 , neighbor.Y )) ||
    (walkable(neighbor.X -1, neighbor.Y + dy)&&
    !walkable(neighbor.X -1, neighbor.Y ))) { return neighbor}
  }}
return makeJump(getVertex(neighbor.X +dx, neighbox.Y + dy), neighbor, goals));}
```

The function “walkable()” from Pseudocode 8 returns true if a vertex is achievable and false in the opposite case. The function “get Neighbors()” returns a list of current vertex neighbors. The performance of the algorithm is presented in the next chapter. The JPS algorithm does not guarantee to find an optimal solution, but at the same time is much faster than A-star or Dijkstra algorithm. Method “createPath()” calculates accumulated radiation from a current vertex to a nonDVertex vertex. The exact implementation depends on a grid implementation and varies from case to case, our case is presented in Pseudocode 9.

Pseudocode 9: Jump Point Search with accumulated radiation, createPath()

```
createPath(current, nonDVertex){
 dx = current.X – nonDVertex.X;
dy = current.Y – nonDVertex.Y;
prev = nonDVertex;
if (dx!= 0 && dy!=0){
        if(|dx| > |dy|){
        slope = dy/dx;
        if (dx>0){
                yTemp = nonDVertex.Y;
                while(i < |dx|){
                if(walkable(current.X +1, yTemp + slope)){
                accumulated += getRadiation(prev, getVertex(prev.X + 1, yTemp + slope))
                prev = getVertex(prev.X + 1, yTemp + slope);
                i++;
                yTemp += slope;}
                }}
        else if (dx<0){
                yTemp = nonDVertex.Y;
                while(i < |dx|){
                if(walkable(current.X -1, yTemp - slope)){
                accumulated += getRadiation(prev, getVertex(prev.X - 1, yTemp - slope));
                prev = getVertex(prev.X - 1, yTemp - slope);
                i++;
                yTemp -= slope;}
                }}
        }else {
                slope = dx/dy;
                if (dy>0){
                xTemp = nonDVertex.X;
                while(i < |dy|){
                if(walkable(xTemp + slope, prev.Y + 1)){
                accumulated += getRadiation(prev, getVertex(xTemp + slope, prev.Y + 1));
                prev = getVertex(xTemp + slope, prev.Y +1);
                i++;
                xTemp += slope;}
                }}
        } else
```

```
    if (dy < 0){
        xTemp = nonDVertex.X;
        while(i < |dy|){
        if(walkable(xTemp - slope, prev.Y - 1)){
        accumulated += getRadiation(prev, getVertex(xTemp - slope, prev.Y - 1));
        prev = getVertex(xTemp - slope, prev.Y -1);
        i++;
        xTemp -= slope;}
        }}
    }
} else
if (dx=0){
    if(dy>0){
        while(i < |dy|){
        if(walkable(prev.X, prev.Y+1)){
        accumulated +=getRadiation(prev, getVertex(prev.X, prev.Y+1));
        prev = getVertex(prev.X, prev.Y + 1);
        i++;
    }}
    }else{
        while(i < |dy|){
        if(walkable(prev.X, prev.Y-1)){
        accumulated +=getRadiation(prev, getVertex(prev.X, prev.Y - 1));
        prev = getVertex(prev.X, prev.Y - 1);
        i++;
    }}}
    }else {
    if(dx>0){
        while(i < |dx|){
        if(walkable(prev.X+1, prev.Y)){
        accumulated +=getRadiation(prev, getVertex(prev.X+1, prev.Y));
        prev = getVertex(prev.X + 1, prev.Y);
        i++;
    }}
    }else{
        while(i < |dx|){
        if(walkable(prev.X-1, prev.Y)){
        accumulated +=getRadiation(prev, getVertex(prev.X-1, prev.Y ));
        prev = getVertex(prev.X-1, prev.Y);
        i++;
    }}}}
return accumulated;
```

6.4 Traveling Salesman Problem

The operational task could be described as visiting several locations within one trip with minimization of risk for the health, in other words, minimization of accumulated radiation. The problem is therefore similar to the classic Traveling Salesman problem, where the salesman is required to visit all necessary locations, only once, with minimization of distance. For testing purposes, we have decided to add such a modification to our program to create graph maps for the TSP problem. The user works with the same interface and functions, the only difference is a lifted limitation on the number of end vertices. In the launcher user choose one method among those described before and launch the search as usual. The application forms distance and radiation matrices launching the pathfinding search in different threads. After the matrices has been generated, the Gurobi package solves three TSP problems, optimization according to accumulated radiation, distance, and both of them using the Sum-Weight method.

The mathematical model is almost identical to that is described in Chapter 4.5 but has two additional sub-cycle eliminating constraints, which are named Miller, Tucker, Zemlin constraints (MTZ) and a modified objective function:

$$Minimize \sum_{i=1}^{n}\sum_{j=1}^{n} c_{ij}x_{ij}\, w_2 + \sum_{i=1}^{n}\sum_{j=1}^{n} r_{ij}x_{ij}w_1 \tag{28}$$

$$w_1 + w_2 = 1$$

$$1 \le u_i \le n-1, \forall i \in N\backslash\{s\}$$

$$u_j \ge u_i + 1 + (n-1)\big(x_{ij} - 1\big), \forall i,j \in N\backslash\{s\}, i \ne j$$

where,

- u_i – are continuous variables interpreted as the number of units of load after visiting vertex i, (it is assumed that each time a vertex is visited one unit of load is collected)
- N – is a set of vertices,
- s – is a starting vertex,
- c_{ij} – is a distance between a vertex i and j, $\forall i,j \in N$,

- r_{ij} $-$ in an accumulated radiation between vertices i and j, $\forall i, j \ \in N$,
- x_{ij} $-$ is a binary variable, where 1 means a route between i and j is used, $\forall i, j \ \in N$,
- w_1 $-$ weight of radiation,
- w_2 $-$ weight of distance.

7. Findings

In this chapter, we present test instances created in the redactor to compare the performance of the algorithms. First, we test the performance of the algorithms without radiation contamination to see the variance in the elapsed time. Secondly, we check the performance of the modified algorithm for accumulated radiation. The testing set consists of 6 different instances aimed to highlight the advantages and flaws of the tested algorithms. All the tests are executed 10 times and the average time is considered. Section 7.1 presents the computational results, comparing the three algorithms and Section 7.2 contains a discussion about the performance.

7.1 Computational results

All the tests are performed on a computer with specifications presented in Table 7.1:

Processor	Apple M1 Pro
Number of cores	10
Memory (RAM)	16
Operating system	Mac OS Monterey 12.3.1

Table 7.1 Table of computer specifications.

Instance 1 (without radiation)

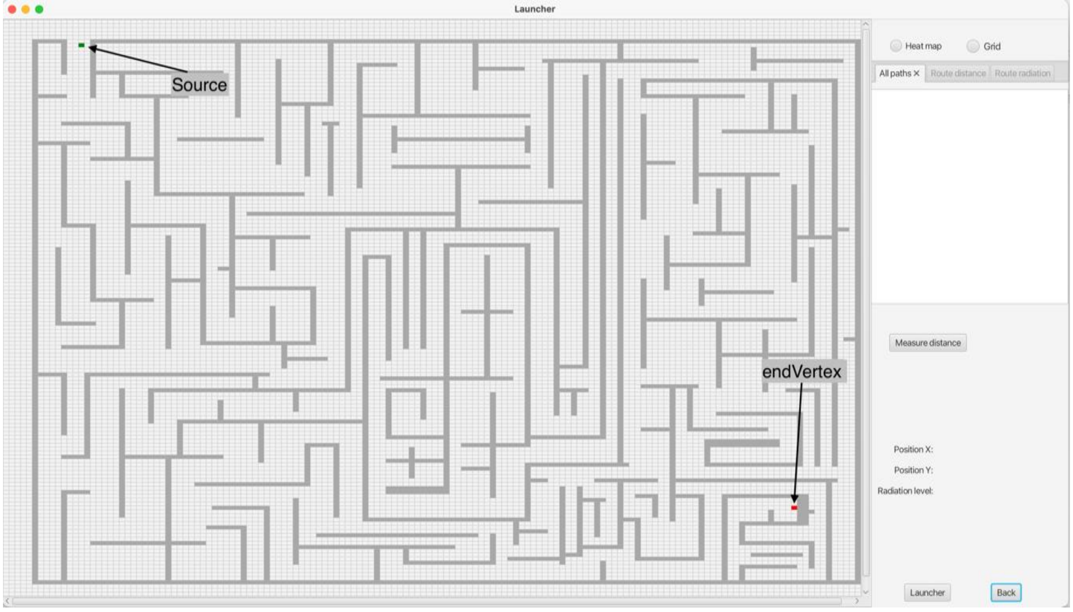
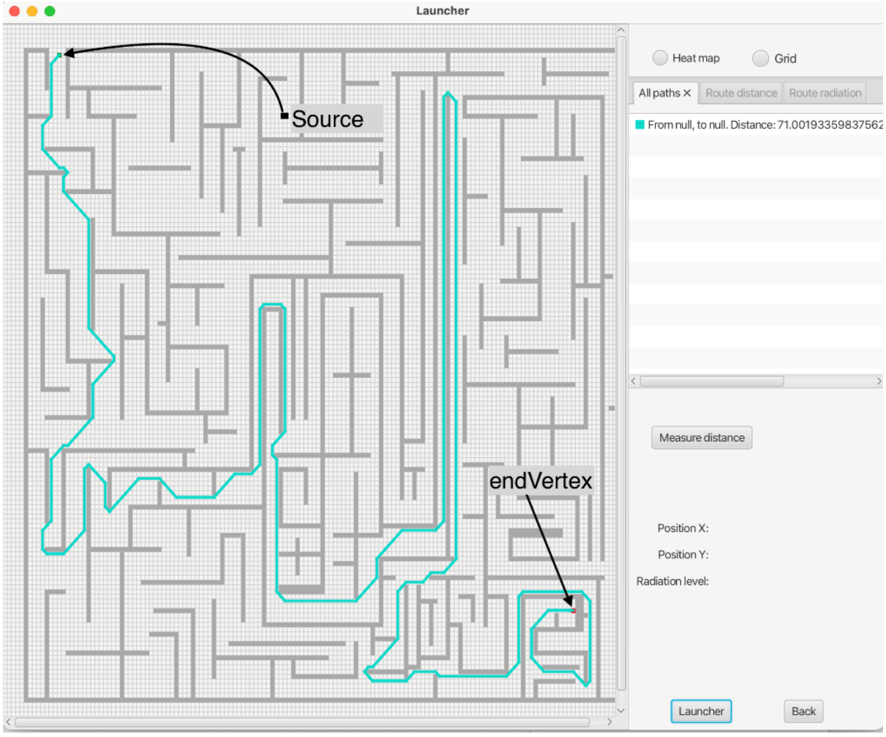


Figure 7.1 Instance 1 as seen from the Launcher.

Figure 7.1 illustrates Instance 1 as seen from the Launcher in the RadPathFinder software. The green cell on the grid is the source (start point), the red cell is the end point. The dark grey cells present obstacles that are not traversable. This instance does not contain any radiation sources and the aim is to demonstrate the pure performance of all three algorithm without additional modifications. The instance presents a map of size 15 × 15 meters and consists of 22 500 vertices. Figure 7.2 below illustrates the optimal path presented by a line of aqua colour. The solution is the same for all three algorithms and the only difference is in elapsed time.



🟦 From null, to null. Distance: 71.00193359837562

Figure 7.2 Optimal path of Instance 1 found by all three algorithms tested.

Algorithm	Elapsed time, sec	Distance, m	Accumulated radiation, mSv/h	Distance gap from optimal, %	Accumulated radiation gap from optimal, %	Elapsed time gap from the best, %
Dijkstra	0.0373	71.0019	0	0%	0%	648.80%
A-star	0.1150	71.0019	0	0%	0%	2211.52%
JPS	0.0050	71.0019	0	0%	0%	0%

Table 7.2 Performance results of the algorithms tested on Instance 1.

From Table 7.2, we can see that the JPS algorithm is much faster than the others. From the elapsed time gap column, we see that A-star algorithm has a gap of 2211.52 %, while the Dijkstra algorithm has a time gap of 648.80 % compared to the JPS method. It might be a bit surprising that A-star performs around 3 times worse than Dijkstra, but this can be explained. The heuristic used in the A-star algorithm aims to direct the exploration of a grid and does not benefit from a large number of obstacles on the way to the endpoint. Moreover, the operations of adding and removing from open and closed lists take a considerable amount of time and cause performance degradation. At the same time, the A-star algorithm performs better than Dijkstra in cases when there is a direct path or just a few obstacles on the way to the endpoint.

Instance 2

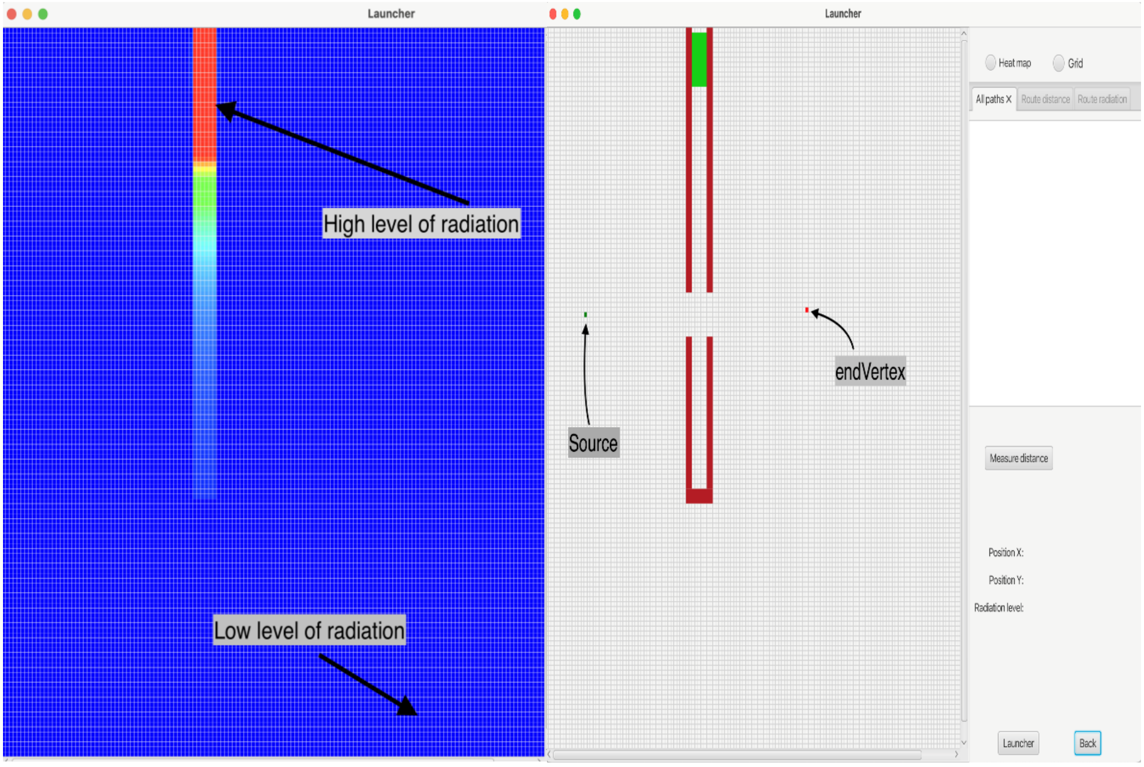
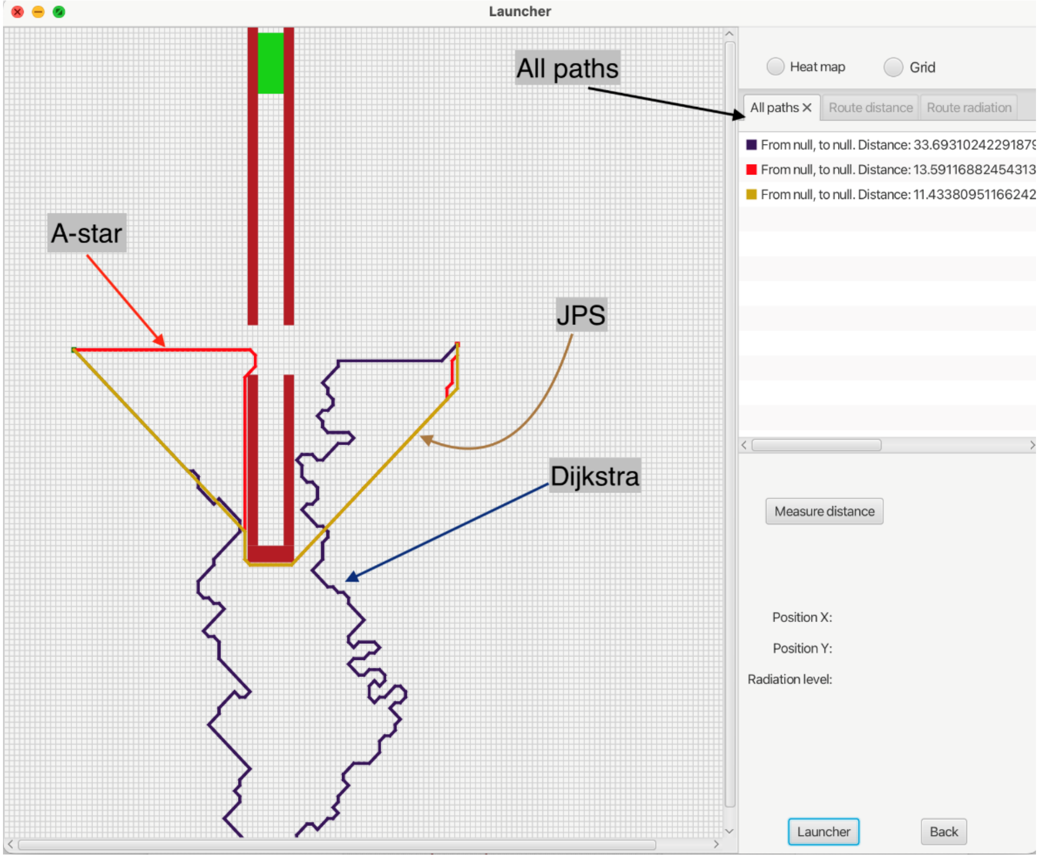


Figure 7.3 Instance 2 with radiation heat map to the left.

Instance 2 presented in Figure 7.3 might seem at first sight as a quite simple case, where the radiation source is surrounded by lead shields that have a break allowing a direct path between source and endpoint. The tricky moment is that there is an obvious direct path between the points, but following it means the accumulation of a high level of radiation, which the algorithms should avoid. The instance size is 15 × 15 meters with 22 500 vertices.

The shortest path for this instance is 7.44 meters with accumulated radiation of 0.0043 mSv. The solutions are presented graphically in Figure 7.4 below, where the path in red colour presents the solution of A-star algorithm, the path found by Dijkstra algorithm coloured blue, and the JPS path is shown in sand.



■ From null, to null. Distance: 33.69310242291879

■ From null, to null. Distance: 13.59116882454313

■ From null, to null. Distance: 11.43380951166242

Figure 7.4 Paths obtained with the different algorithms.

Algorithm	Elapsed time, s	Distance, m	Accumulated radiation mSv/h	Distance gap from optimal, %	Accumulated radiation gap from optimal, %	Elapsed time gap from the best, %
Dijkstra	0.0446	33.6931	1.74304E-11	352.86%	0.00%	430.96%
A-star	0.0654	13.4740	2.94E-11	81.10%	68.74%	678.19%
JPS	0.0084	11.4338	2.97528E-11	53.68%	70.69%	0.00%

Table 7.3 Performance results of the algorithms tested on Instance 2.

From Table 7.3 we observe that our modified JPS finds the solution around 6 times faster than the rivals, but the accumulated radiation is 70 % worse than the optimal.

At the same time, the optimal path cannot be repeated in high precision by a human due to an extremely large number of turns, which makes it unreproducible. JPS provides the shortest path among those proposed by other algorithms for radiation avoidance. The results of the A-star algorithm presented in table 3 are taken for the radiation and distance wights equal 0.5. It gives intermediate results between Dijkstra and JPS but is the slowest. The pareto front for Instance 2 is shown graphically below in Figure 7.5.

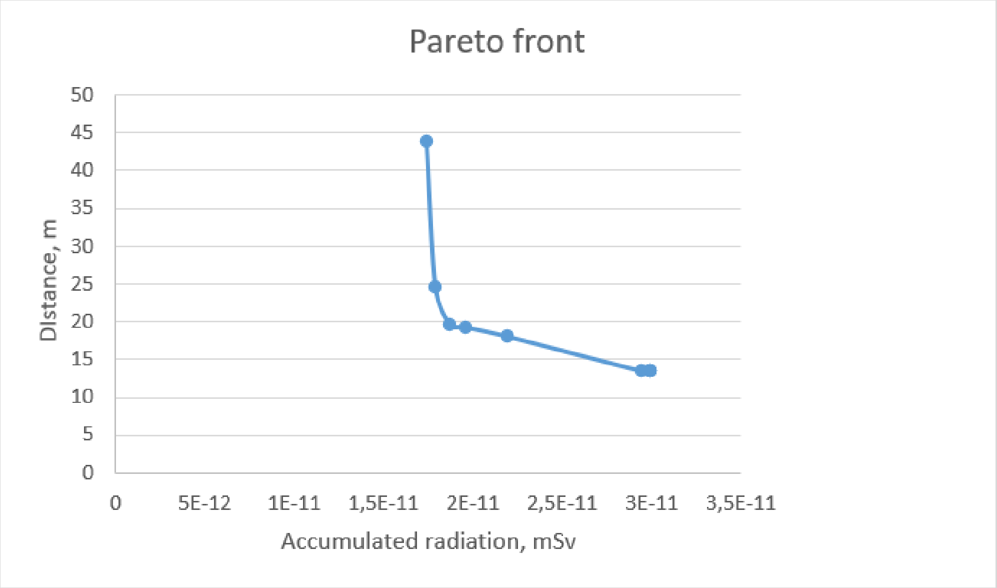


Figure 7.5 Pareto front for Instance 2.

Instance 3

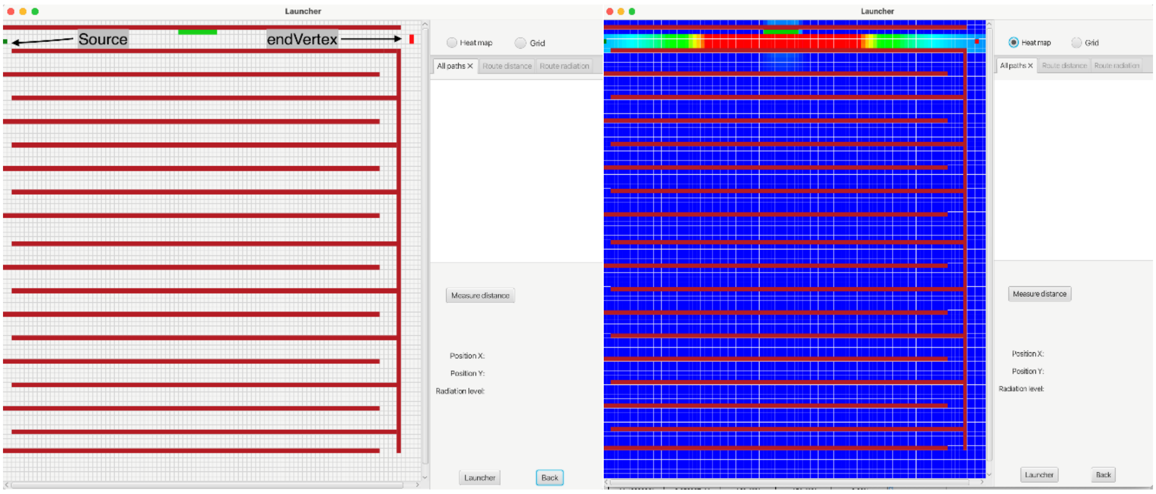


Figure 7.6 Instance 3 with heatmap-view to the right.

Instance 3 is shown in Figure 7.6 above and has a similar idea as the second, there is a direct and short path between the source and the endpoint.

However, even though the path is considerable short, there is a highly contaminated zone along it, which cause a high level of accumulated radiation for a person or drone. This instance helps us to ensure that the algorithms try to avoid regions with high radiation. The instance size is 10 × 10 meters with 10 000 vertices. The shortest distance between the source and endpoint is 9.5 meters and accumulated radiation is 5.5045 mSv. Figure 7.7 below shows the path found with the JPS method.

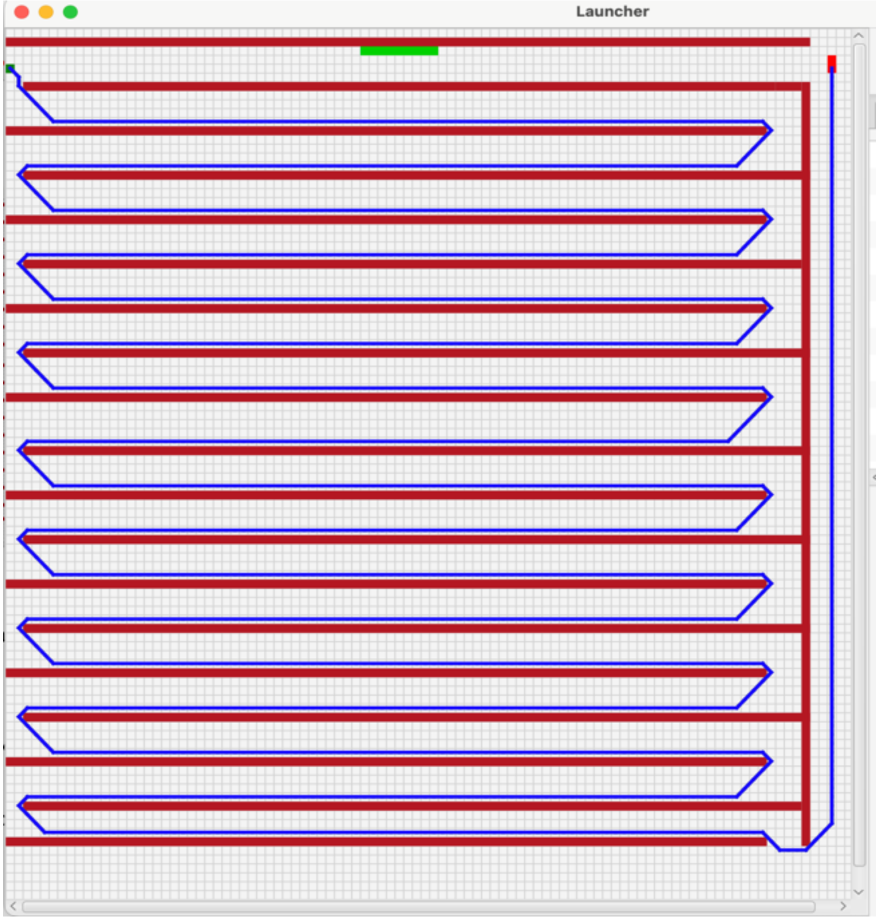


Figure 7.7 Path found with JPS method for Instance 3.

Algorithm	Elapsed time, s	Distance, m
Dijkstra	0.0201	170.9583
A-star	0.0492	166.5007
JPS	0.0055	161.0279

Table 7.4 Elapsed time and distance results from Instance 3.

Algorithm	Accumulated radiation mSv/h	Distance gap from optimal, %	Accumulated radiation gap from optimal, %	Elapsed time gap from the best, %	Dist. gap from the best among heuristics, %
Dijkstra	0.0069	1699.56%	0%	267.70%	6.17%
A-star	0.0074	1652.64%	6.72%	802.16%	3.40%
JPS	0.0075	1595.03%	7.52%	0%	0%

Table 7.5 Performance results of the algorithms tested on Instance 3.

For this instance, the modified JPS outperforms the other algorithms at least three times when it comes to elapsed time, as can be seen in Table 7.4. From Table 7.5 above, we see that the accumulated radiation is just slightly worse for the JPS compared to the optimal accumulated radiation, which is negligible if we look at absolute numbers. The performance of the A-star algorithm can be explained by a high number of obstacles. The instance does not present any interest in other aspects and serves only for verification purposes.

Instance 4

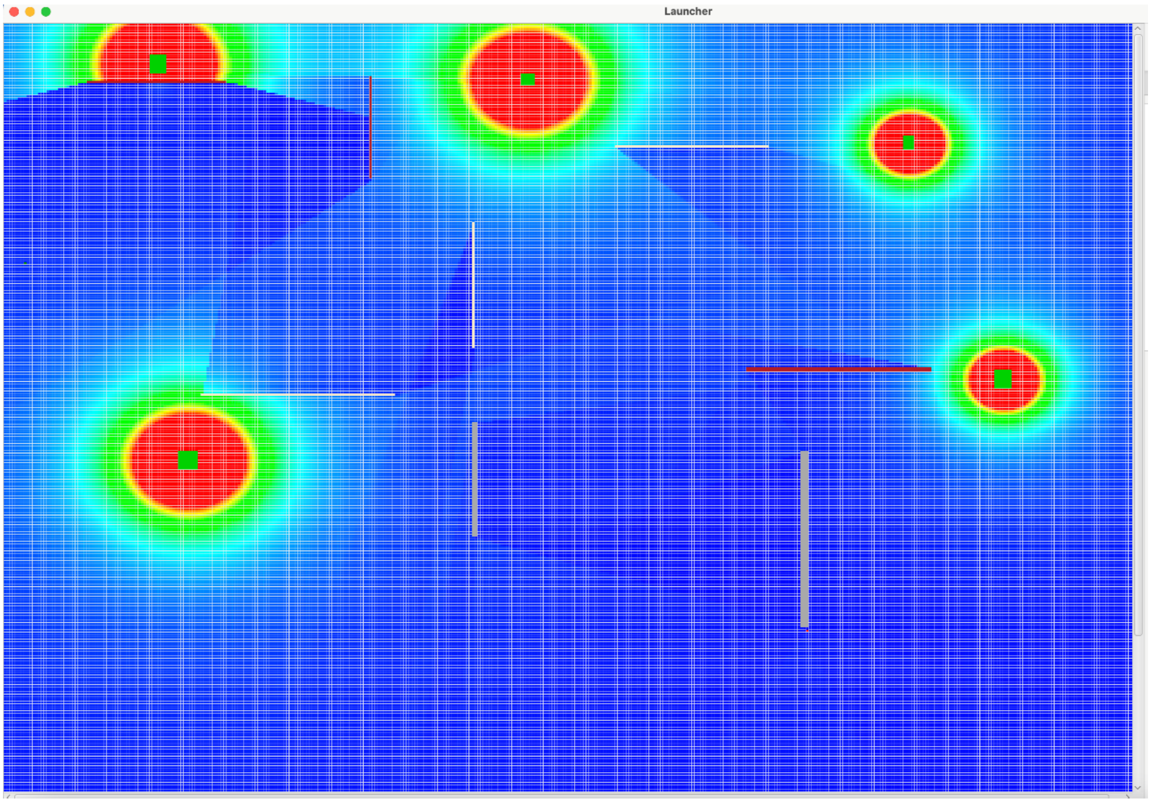
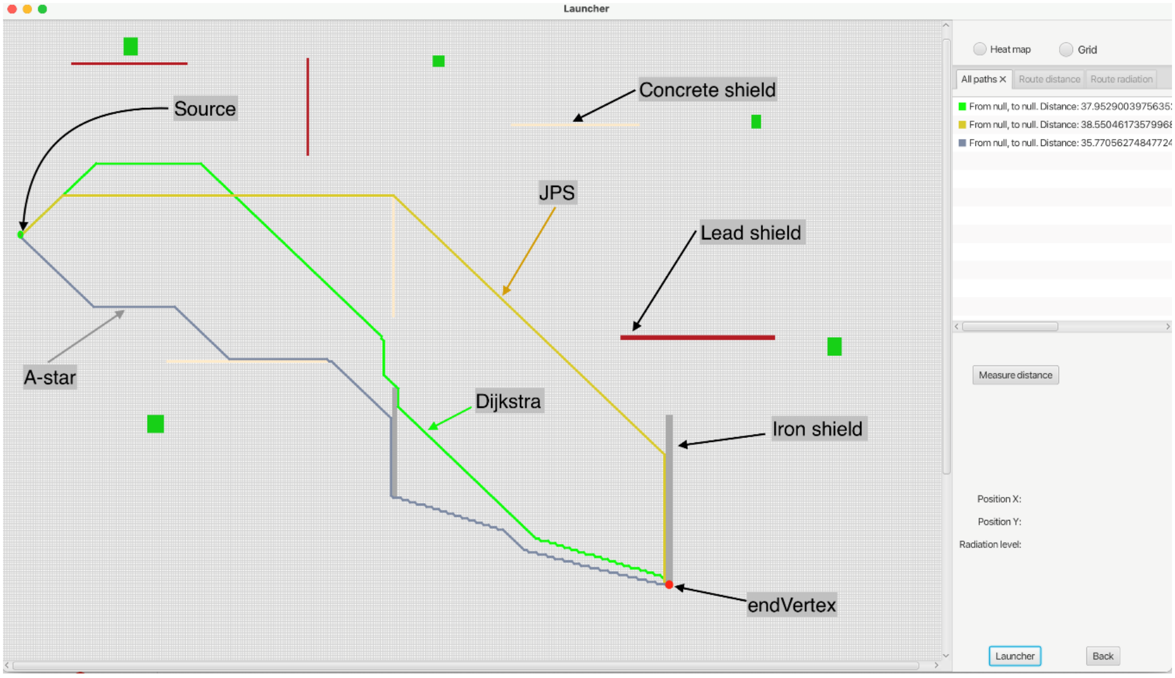


Figure 7.8 Instance 4 with several radiation sources.

The instance presents a case with radiation sources on all sides of the map as can be seen in Figure 7.8. A few shields are allocated in a way to reduce the harmfulness effect of radiation. The number of obstacles is not high; therefore, we can expect a good performance from the A-star algorithm. The size of instance is 40 × 40 meters with 160 000 vertices.



🟩 From null, to null. Distance: 37.9529003975635
🟨 From null, to null. Distance: 38.5504617357996
■ From null, to null. Distance: 35.7705627484772

Figure 7.9 Paths found with the three algorithms for Instance 4.

From Figure 7.9 above the paths found for Instance 4 is presented. The green line presents an optimal solution according to accumulated radiation found by Dijkstra, the grey line is a path founded by the A-star algorithm, and the line of a sandy colour is the solution obtained with JPS. The weights used for the A-star, presented in this figure, are equal to 0.5.

The optimal path according to the distance has following characteristics, the distance is equal to 33.77 meters, with the accumulated radiation equal to 0.2110 mSv. The optimal path according to the radiation is following, the distance is 37.95 meters, the radiation is 0.0577 mSv. Table 7.6 below shows the elapsed time in seconds and distance in meters.

Algorithm	Elapsed time, s	Distance, m
Dijkstra	0.1429	37.9529
A-star	0.0088	35.7706
JPS	0.0521	38.5505

Table 7.6 Elapsed time and distance results from Instance 4.

Algorithm	Accumulated radiation mSv/h	Distance gap from optimal, %	Accumulated radiation gap from optimal, %	Elapsed time gap from the best, %	Distance gap from the best among heuristics, %
Dijkstra	0.0578	12.36%	0%	1514.63%	6.10%
A-star	0.1072	5.90%	85.67%	0%	0%
JPS	0.0942	14.13%	63.06%	488.62%	7.77%

Table 7.7 Performance results of the algorithms tested on Instance 4.

From Table 7.7 we can see that the performance of the A-star algorithm is quite impressive for this instance. It leaves behind all other algorithms and finds the solution 16 times faster than Dijkstra and almost 6 times faster than JPS. However, these numbers are quite ambiguous. The reader should remember that a correct implementation of the Sum-Weighted method requires normalization, which means there is a need in computing the optimal distance and accumulated radiation dose. From this point, the JPS becomes the fastest algorithm for this instance as well. The accumulated radiation for JPS and A-star algorithms are similar with a small vantage of JPS. The pareto front for Instance 4 is shown below in Figure 7.10.

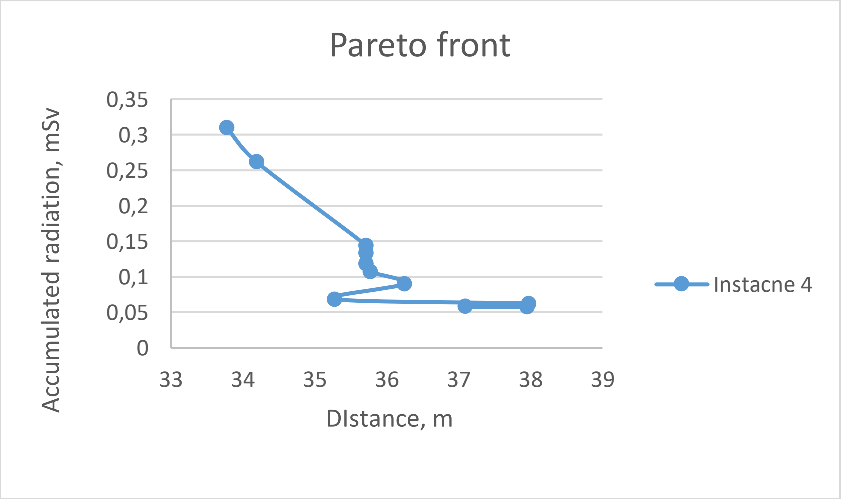


Figure 7.10 Pareto front for Instance 4.

Instance 5

Instance 5 is the largest in our testing pool. It represents an area of 100x100 meters with 1 000 000 vertices. The radiation sources (as seen in Figure 7.11) and shields of various materials are randomly scattered on the map. The instance is suitable to test the performance of the algorithms as it has comparably high number of vertices. The number of vertices significantly affects the performance of Dijkstra algorithm; therefore, we can expect the longest elapsed time.

The optimal path in term of distance is 120.97 meters with accumulated radiation 0.183 mSv. The optimal path according to accumulated radiation has following parameters, the distance is 308.5 meters and the accumulated radiation 0.0104. Figure 7.12 illustrates the paths found with the three algorithms on this instance. The line of aqua colour shows the solution of the A-star algorithm, the blue colour shows the path produced by Dijkstra, and the green line belongs to JPS.

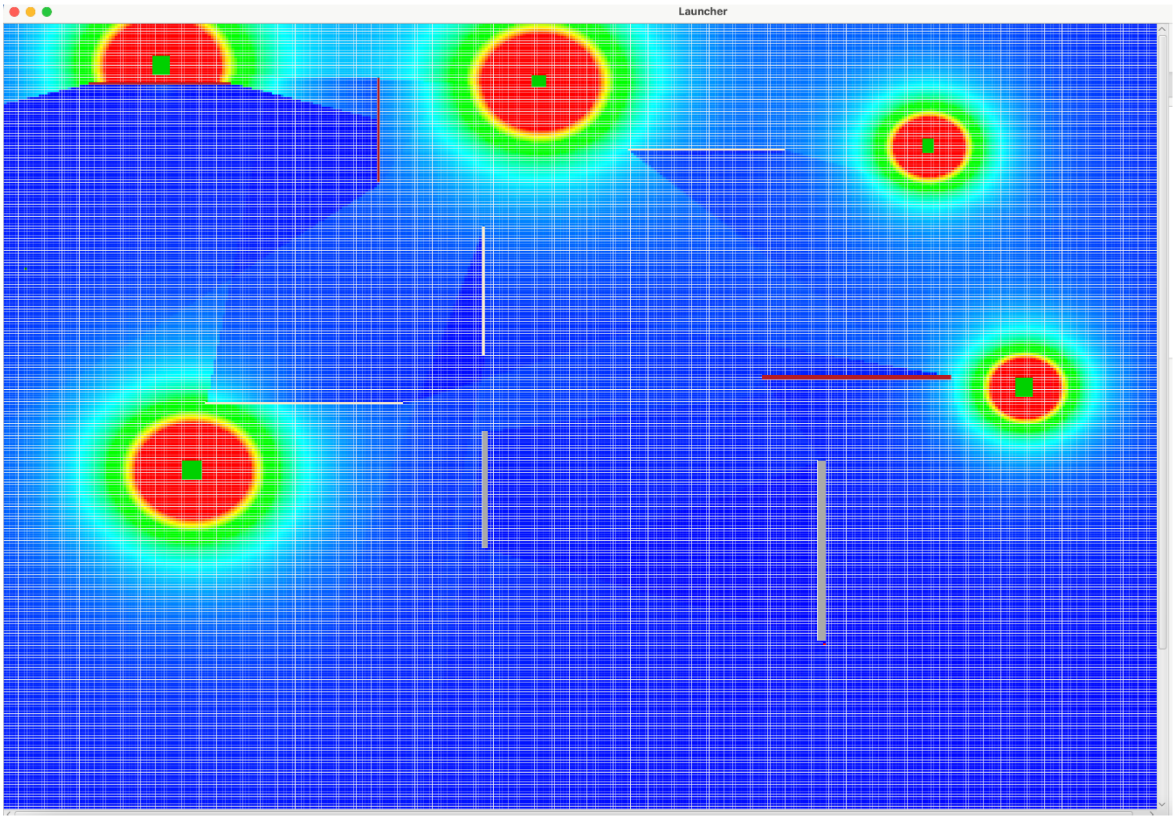


Figure 7.11 Instance 5 with heatmap-view.

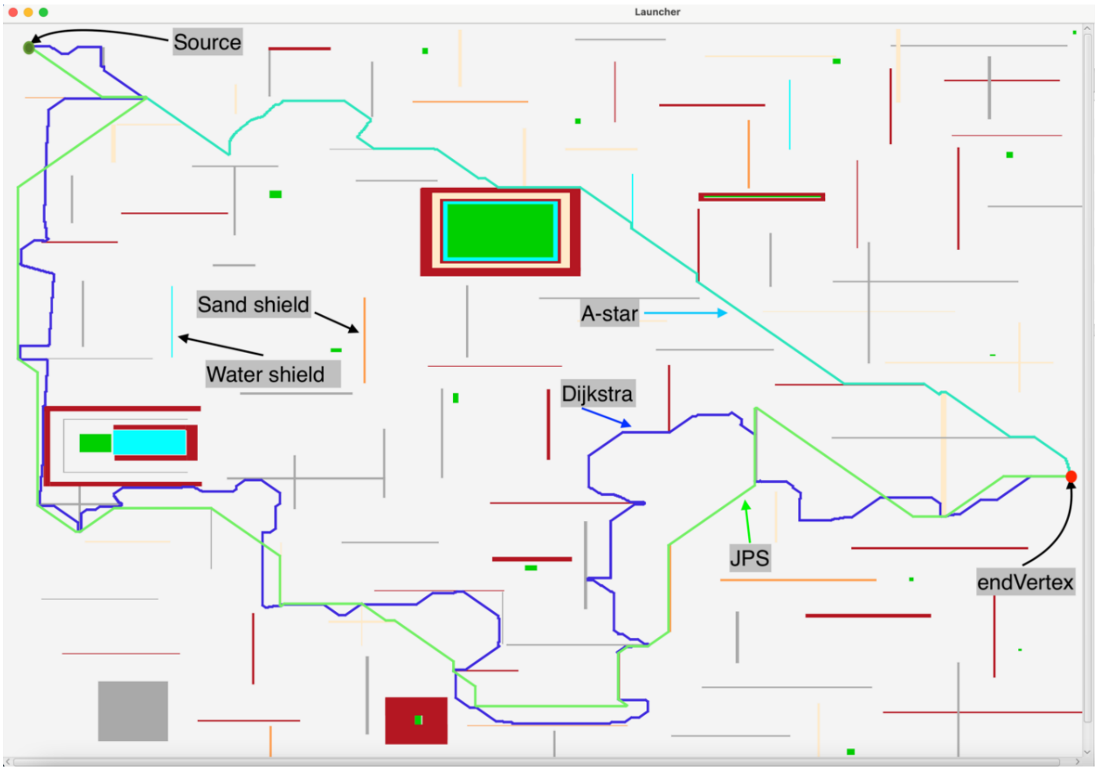


Figure 7.12 Paths found with the three algorithms for Instance 5.

Algorithm	Elapsed time, s	Distance, m
Dijkstra	3.7374	308.5180
A-star	1.7182	132.0854
JPS	0.3178	229.6418

Table 7.8 Elapsed time and distance results from Instance 5.

Algorithm	Accumulated radiation mSv/h	Distance gap from optimal, %	Accumulated radiation gap from optimal, %	Elapsed time gap from the best, %	Distance gap from the best among heuristics, %
Dijkstra	0.0105	155.02%	0%	1076.06%	133.57%
A-star	0.0316	9.18%	202.16%	440.67%	0%
JPS	0.0141	89.82%	34.76%	0%	73.86%

Table 7.9 Performance results from of the algorithms tested on Instance 5.

From the Tables 7.8 and 7.9 we can see that the performance of the JPS method is quite impressive for this instance. It is 11 times faster than the Dijkstra and 5 times faster than A-star algorithm.

The Dijkstra takes almost 4 seconds in average that can be quite a significant drawback in cases when the calculations should be held several times. The path has quite many turns, that makes it difficult to be followed by a human or a drone. The accumulated radiation is slightly higher in a solution produced by JPS but can be assumed as negligible. At the same time, the distance in the solution from JPS is around 80 meters shorter and is much easier to be passed. The A-star provides an intermediate solution without any significant benefits on any objectives. The pareto front for the instance is visualized below in Figure 7.13.

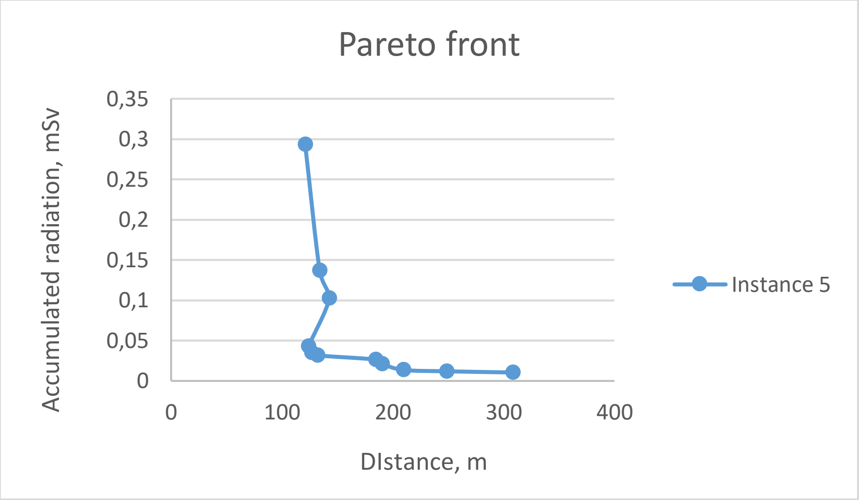
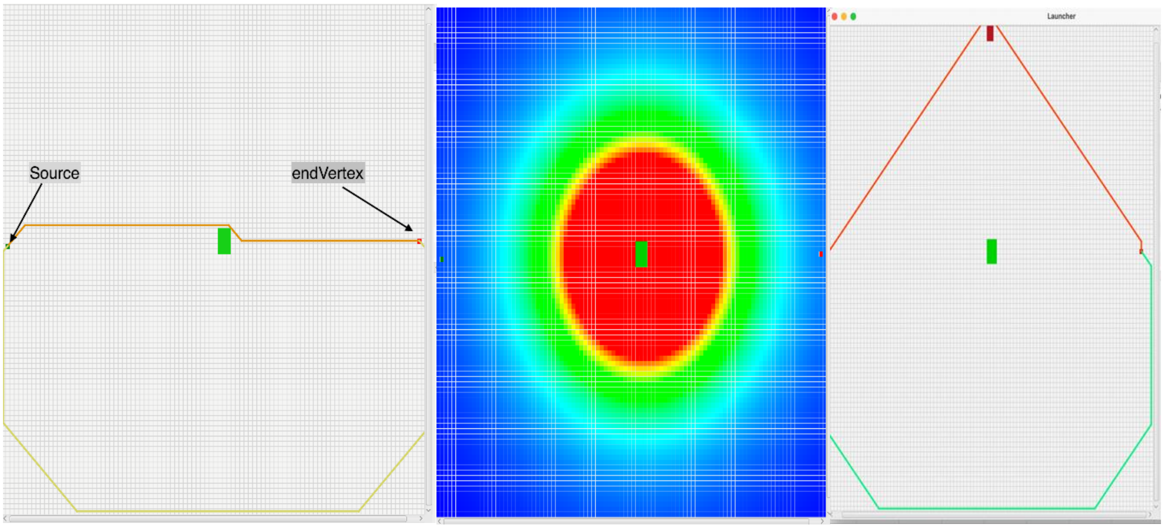


Figure 7.13 Pareto front for Instance 5.

Instance 6



(a) (b) (c)

Figure 7.14 Visualisations of test Instance 6. (a) Paths found with no obstacles (b) Heatmap-view (c) Paths found with one obstacle.

Instance 6, as shown in Figure 7.14, is a demonstration of the main drawback of the modified Jump Point Search algorithm. Figure 7.14(a) is the initial version of the instance which has no shields and only a radiation source. The solution of JPS is presented with a brown line, which goes along the radiation source. Obviously, such a solution is far from optimal. The line of a yellow colour shows the path produced by the Dijkstra algorithm. The accumulated radiation for the Dijkstra is 0.12 mSv, while for the JPS is 2.78 mSv.

The problem is hidden in the fundamental idea of the algorithm. To perform a proper search in a radiation environment the JPS needs a few non-dominated vertices to compute the accumulated radiation between the points and choose the best combination of them. When there are no obstacles on a grid map, only a few non-dominated vertices are found around the end vertex. For this reason, the usage of the proposed modification of the JPS is not recommended for any instances without shields. At the same time after adding one obstacle the results produced by the JPS is incomparably better as shown in Figure 7.14(c). The accumulated radiation for the JPS is 0.1842 mSv, which is 15 times better than the previous one. The solution of the Dijkstra algorithm stays the same.

Instance 7

Instance 7 consists of several shields and radiation sources and is not aimed to present any unique features of the algorithms. The primary purpose of the instance is to show the performance of the algorithms with the growth of the instance's size. The testing set is presented with the maps of sizes 2×2, 10×10, 30×30, and 100×100 meters. The cases are identical in dislocation of the shields and radiation sources but are not guaranteed to be the same in radiation levels of each point. Figure 7.15 below shows the layout of Instance 7.

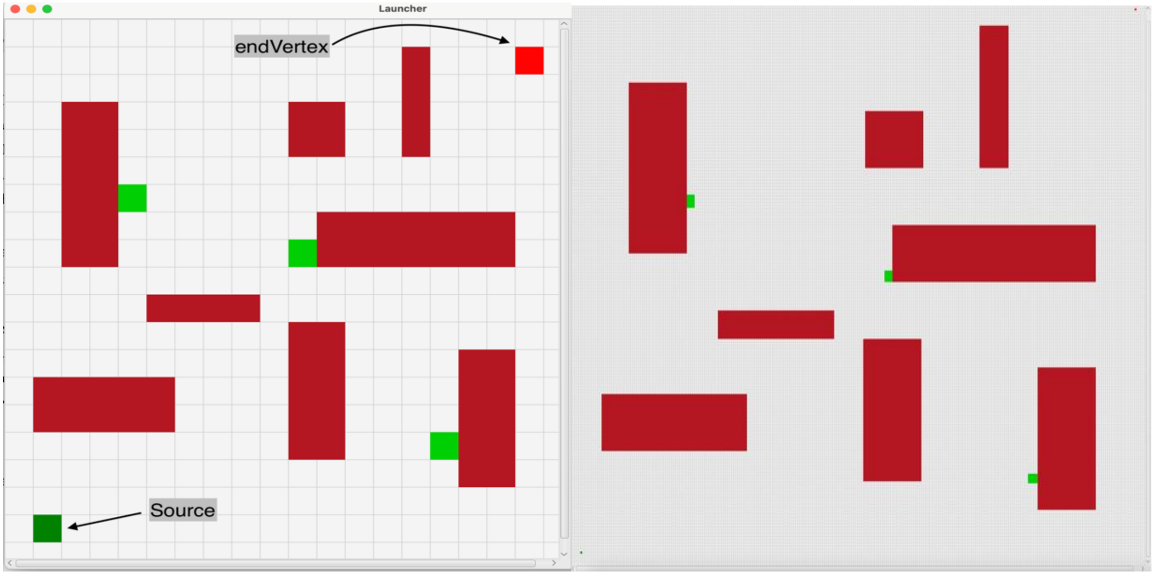


Figure 7.15 Illustration of Instance 7.

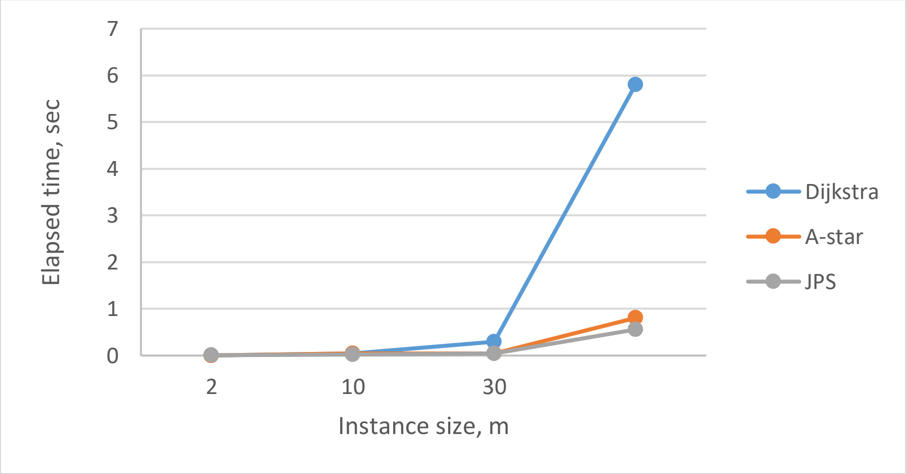


Figure 7.16 Elapsed time for different grid sizes with the three algorithms.

Algorithm/Size	Elapsed time, sec			
	2x2	10x10	30x30	100x100
Dijkstra	0.0030	0.0416	0.2975	5.8034
A-star	0.0036	0.0452	0.0506	0.8056
JPS	0.0067	0.0188	0.0488	0.5587

Table 7.10 Elapsed time in seconds with different grid sizes.

The tests provide a good overview of changes in the performance of the algorithms. From Figure 7.16, the graph shows the elapsed time in seconds for the 4 different grid sizes when using the three algorithms. Table 7.10 presents the same results.

With small grid size instances, the Dijkstra works faster than others but with the growth of the instance size, it becomes much slower. The JPS and A-star show similar growth rates with incrementation of instance size. At the same time, the results highly depend on the topology of maps and can significantly differ for other instances that the reader could witness in the previous instances.

Instance 8 (Traveling Salesman Problem)

Instance 8 aims to show a realistic example of a building with several radiation sources, where a person is required to visit several points and return to the start point. The instance has 6 points that must be visited marked as "endVertex" as shown in Figure 7.17. The instance also includes a high number of obstacles. The instance size is 40 × 30 meters with 120 000 vertices.

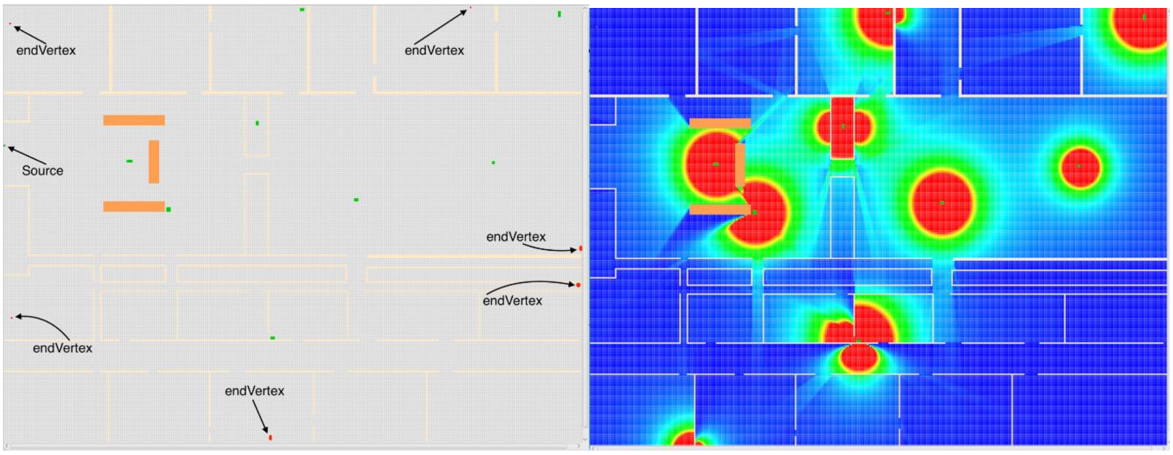


Figure 7.17 Instance 8: Traveling Salesman Problem, with heat-map to the right.

The task can be solved with any of the presented algorithms, but the computational time can differ drastically. The program creates the distance and accumulated radiation matrices between all the vertices to be visited, which means 36 routes for this instance. The TSP problem for the instance is solved with Dijkstra and JPS algorithms. The approaches for solving the task can be different. The user can use classical pathfinding algorithms to find shortest paths between the points and then solve TSP with optimization of accumulated radiation or use the modified algorithms and find the paths with minimized accumulated radiation and only then optimize the total distance. The results for both approaches are presented below.

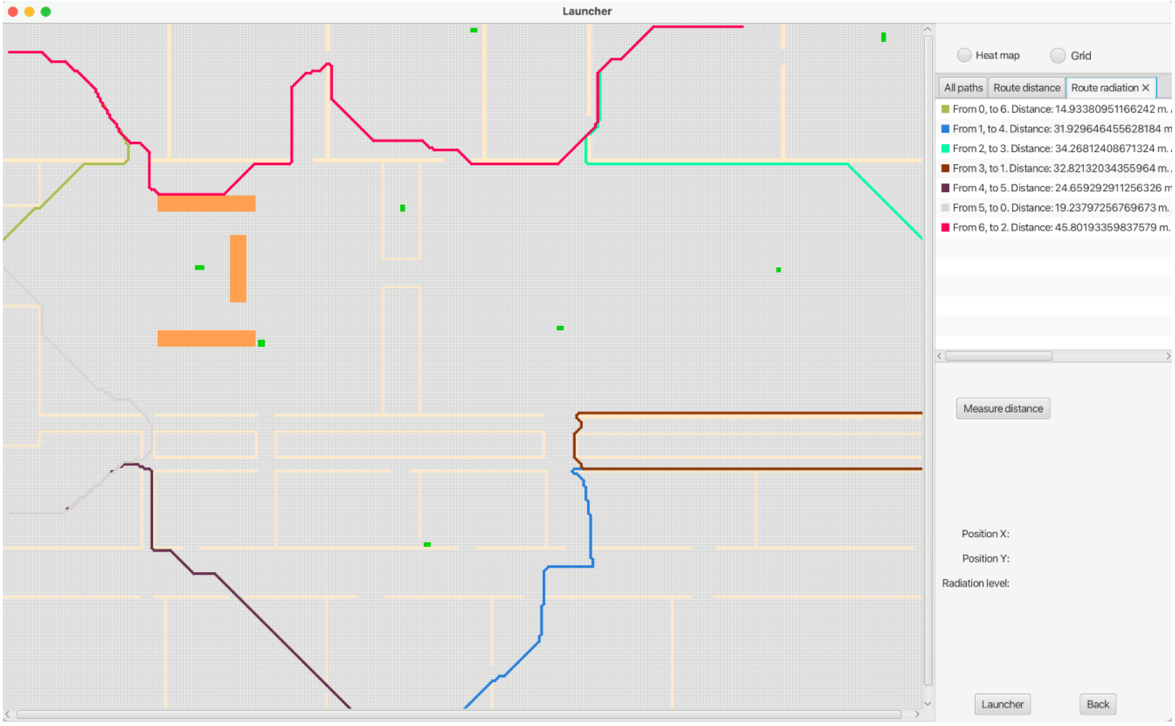
Algorithm	Elapsed time, s	Distance with distance optimization, m	Distance with radiation optimization, mSv	Accum. radiation with radiation optimization, mSv	Accum. radiation with distance optimization, mSv	Distance with weights 0.5, m	Radiation with weights, mSv
Dijkstra	53.4	186.6594	186.6594	0.9342	0.9342	186.6594	0.9342
Dijkstra rad.	53.1	203.6521	203.6521	0.5825	0.5824	203.6521	0.5824
JPS	51.9	186.6594	220.0077	0.9155	10.2843	197.4351	3.5430
JPS rad.	52.1	188.3239	188.3239	0.6827	0.6827	188.3239	0.6827

Table 7.11 Performance results for Instance 8.

Algorithm	Distance with distance optimization, gap from optimal, %	Distance with radiation optimization, gap from optimal, %	Accum. radiation with radiation optimization, gap from optimal, %	Accum. radiation with distance optimization, gap from optimal, %	Distance with weights 0.5, gap from optimal, %	Radiation with weights, gap from optimal, %
Dijkstra	0%	0%	60.40%	60.40%	0%	60.40%
Dijkstra rad.	9.10%	9.10%	0%	0%	9.10%	0%
JPS	0%	17.87%	57.20%	1665.83%	5.77%	508.33%
JPS rad.	0.89%	0.89%	17.22%	17.22%	0.89%	17.22%

Table 7.12 Performance results for Instance 8.

From Table 7.11 and 7.12 above, we can observe the performance results of the TSP problem with the different algorithms tested. It is clear the classical algorithms are not suitable to solve such a problem as the accumulated radiation is too high and create unnecessary risk for human or drones. There is almost no difference between the results of the modified JPS and Dijkstra algorithms. The illustration from Figure 7.18 below shows the path found for the traveling salesman problem. In the next section, we discuss a significant drawback of the Dijkstra algorithm, making the modified JPS method more attractive.



🟩 From 0, to 6. Distance: 14.93380951166242 m.

🟦 From 1, to 4. Distance: 31.929646455628184 m

🟩 From 2, to 3. Distance: 34.26812408671324 m.

🟫 From 3, to 1. Distance: 32.82132034355964 m.

🟪 From 4, to 5. Distance: 24.659292911256326 m

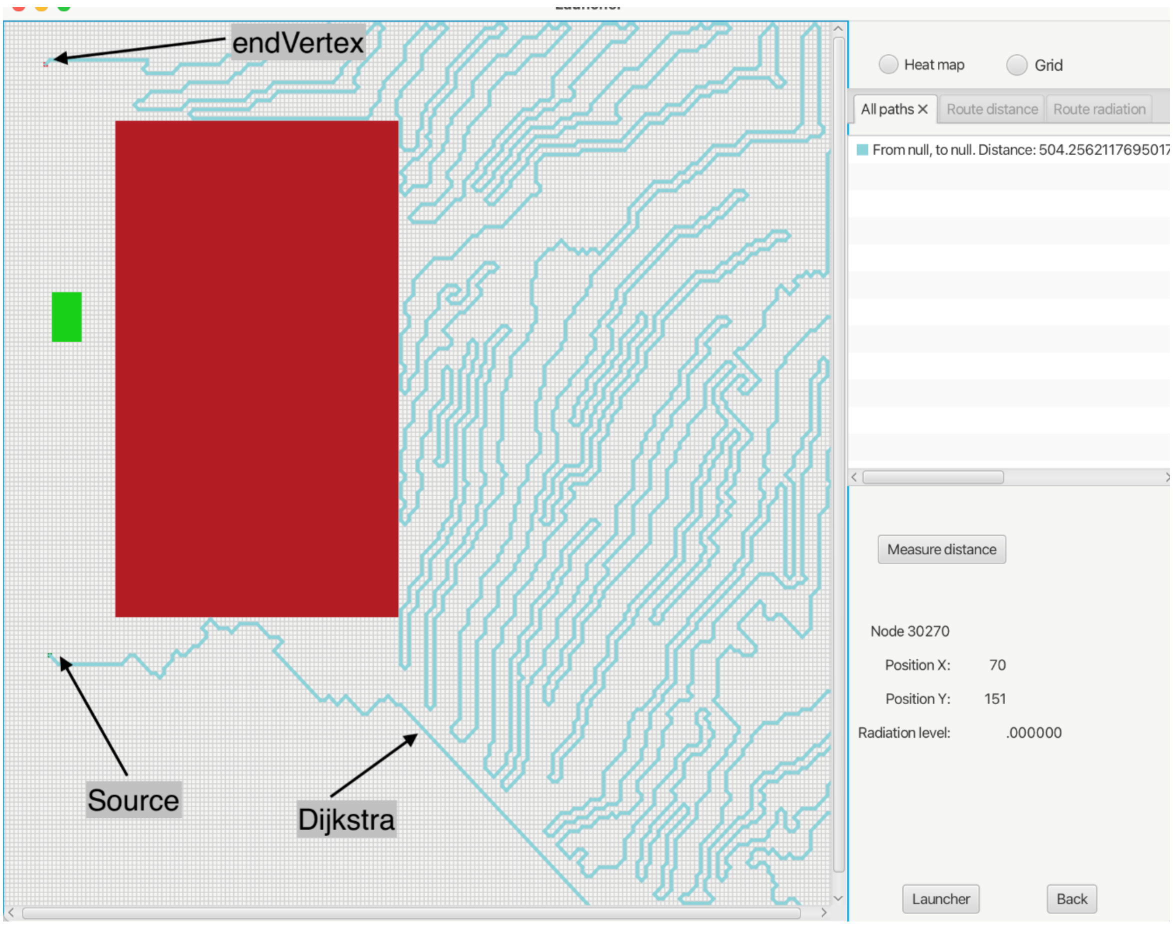
⬜ From 5, to 0. Distance: 19.23797256769673 m.

🟥 From 6, to 2. Distance: 45.80193359837579 m.

Figure 7.18 Path found for the TSP instance.

7.2 Discussion

The Dijkstra algorithm is modified to work with accumulated radiation to find optimal paths and works comparably fast for small and intermediate instances. Unfortunately, it has its drawbacks. In some instances where the radiation is distributed irregularly and there are some areas where the radiation level is equal to zero, the algorithm does not have an instrument that can lead the search in areas without radiation. In such situations, the algorithm chooses between vertices with the same level of accumulated radiation until it finds a vertex with non-zero radiation. In such cases, the elapsed time increases drastically, but the result according to accumulated radiation is still optimal, which is not the case for the distance. The instance presented in Figure 7.19 below is created to demonstrate such a situation.



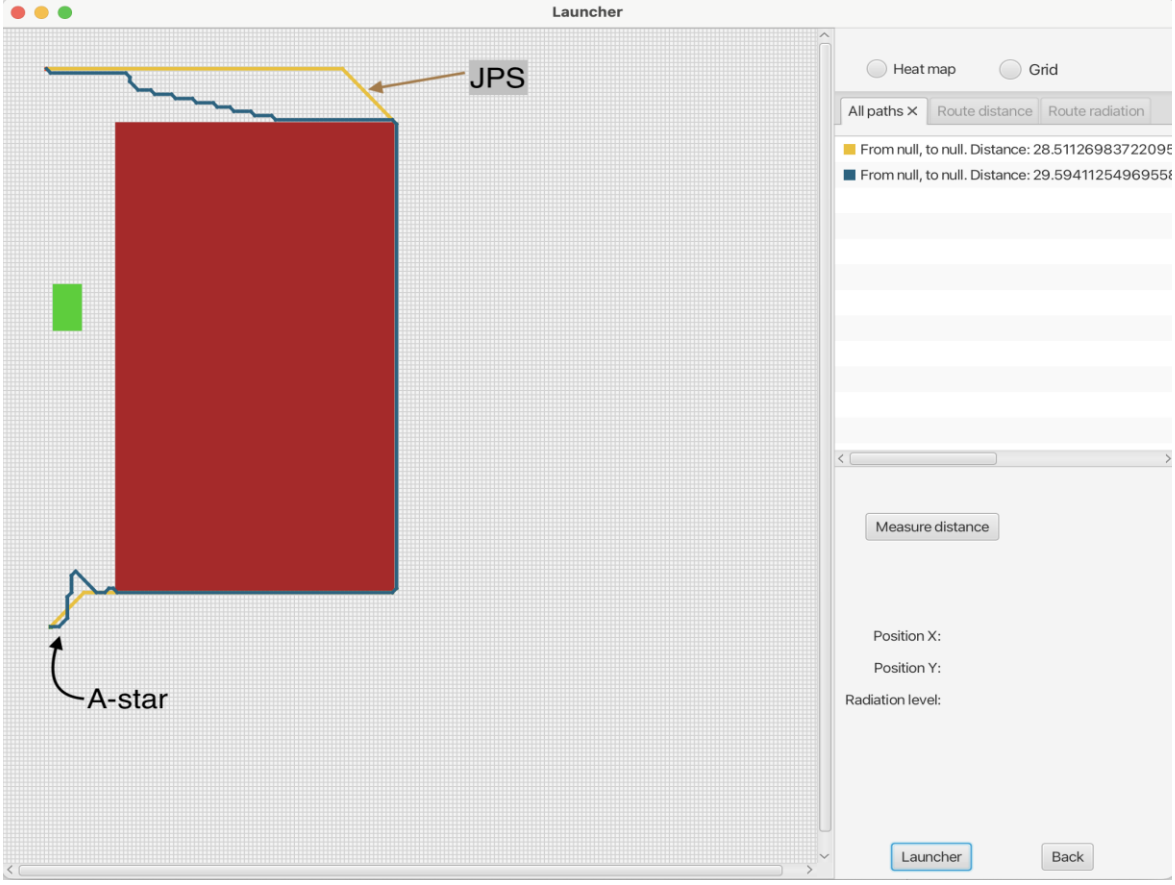
From null, to null. Distance: 504.2562117695017

Node 30270
Position X: 70
Position Y: 151
Radiation level: .000000

Figure 7.19 Test instance for areas without radiation with Dijkstra algorithm.

The instance size is 20 × 20 meters with 40 000 vertices. The source and end point are located in front of each other with a radiation source between them, thus a direct path is not reasonable. On the right side of the radiation source, a lead shield is located. Its thickness equals 10 meters which does not allow the radiation to penetrate throughout. Therefore, there is an area free from radiation, where the algorithm gets stuck. It continues to look for a path, but there is no rule that leads it to the end vertex. As a result, a path as shown in Figure 7.19 is found using the Dijkstra algorithm. The distance of this path is equal to 504 meters, which is completely unreasonable. The accumulated radiation is equal to 0.0192 mSv.

The problem can be solved with the usage of the A-star algorithm or JPS. The A-star traditionally requires two optimal solutions according to distance and accumulated radiation, while the JPS does not need any preparation.



🟨 From null, to null. Distance: 28.51126983722095

🟦 From null, to null. Distance: 29.5941125496955

Figure 7.20 Solutions obtained with JPS and A-star with weights.

From figure 7.20 the found paths with A-star and JPS is presented. The yellow line presents the solution of JPS, the blue line is the result of the A-star with weights equal to 0.5.

Algorithm	Elapsed time, s	Distance, m
Dijkstra	0.4950	504.2562
A-star	0.0265	29.5941
JPS	0.0157	28.5113

Table 7.13 Elapsed time and distance results.

Algorithm	Accumulated radiation mSv/h	Distance gap from optimal, %	Accumulated radiation gap from optimal, %	Elapsed time gap from the best, %
Dijkstra	0.0192	3737.15%	0%	3053.84%
A-star	0.0271	125.20%	40.54%	68.97%
JPS	0.0215	116.96%	11.70%	0%

Table 7.14 Performance results of the three algorithms.

From Tables 7.13 and 7.14, we can see that JPS provides a better solution in all parameters and does not suffer from the flaws of other algorithms. The Dijkstra algorithm with the radiation modification explores the graph differently from the classic version. The distance grows steadily when the algorithm explores the graph, while the radiation intensity can differ on different parts of the graph. Therefore, the search of a path on a graph with radiation can be faster or slower as the algorithm does not explore all the vertices on the way to the endpoint, because some vertices are not reasonable to be explored as they have a large level of radiation. Figure 7.21 demonstrates both cases.

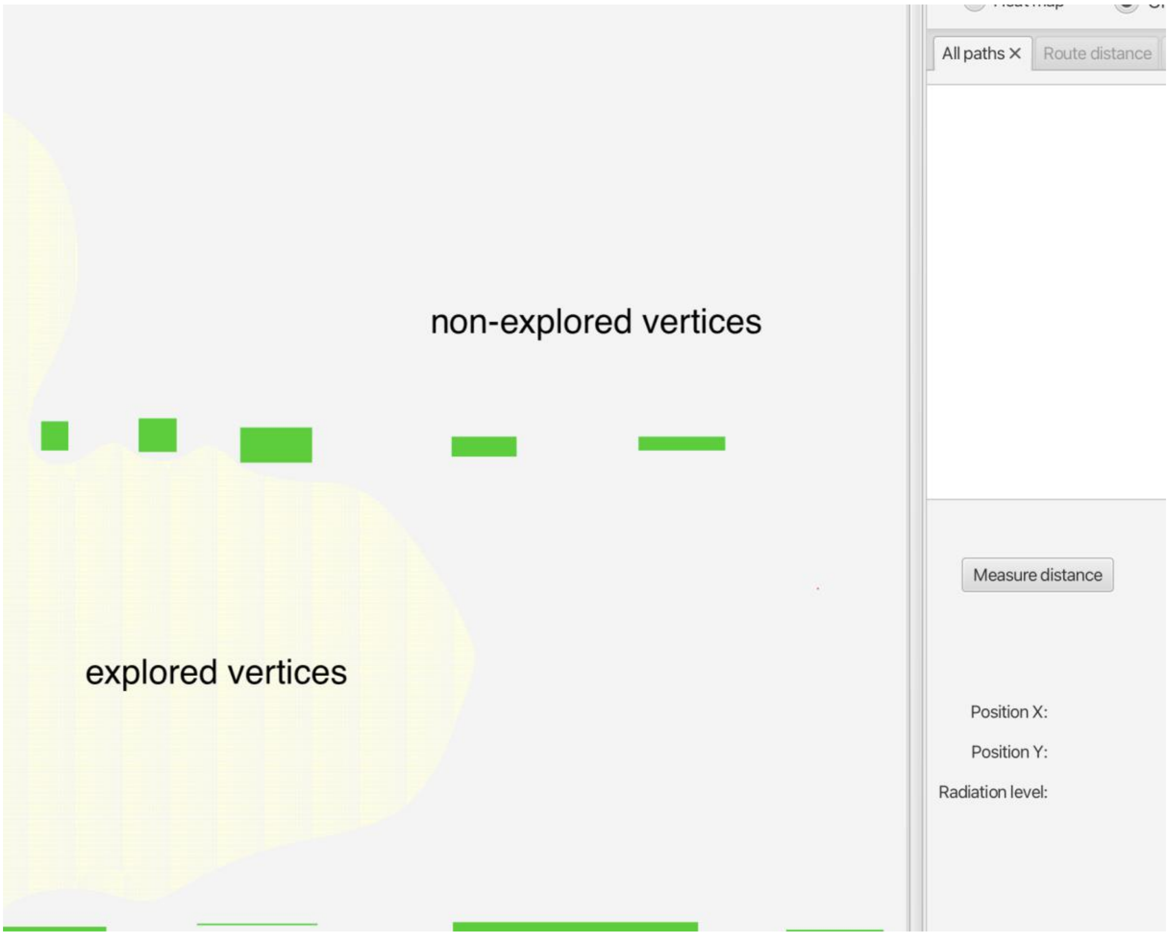


Figure 7.21 Exploration of the grid.

The radiation near the sources is high, therefore the algorithm does not explore the grid steadily. Figure 7.22 below demonstrates how the grid is explored when the algorithms works with distances.

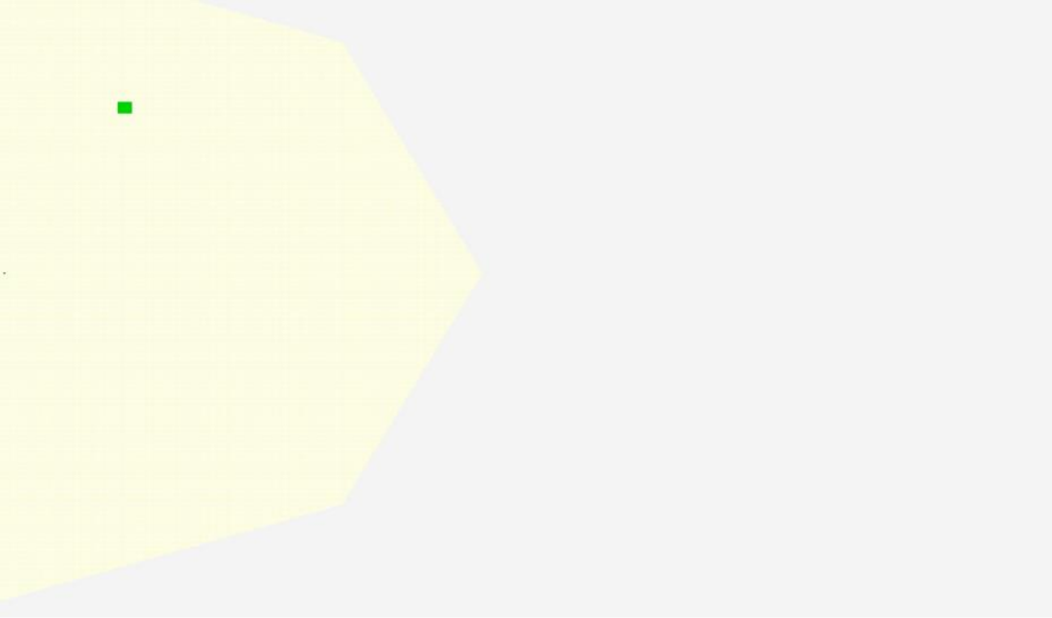


Figure 7.22 Exploration based on distances.

The instance presented in Figure 7.23 shows a case when the search for an optimal path according to accumulated radiation takes a much longer time than the search for the shortest path. The elapsed time for the radiation is 1.3392 sec, while for the distance is only 0.0596 sec.

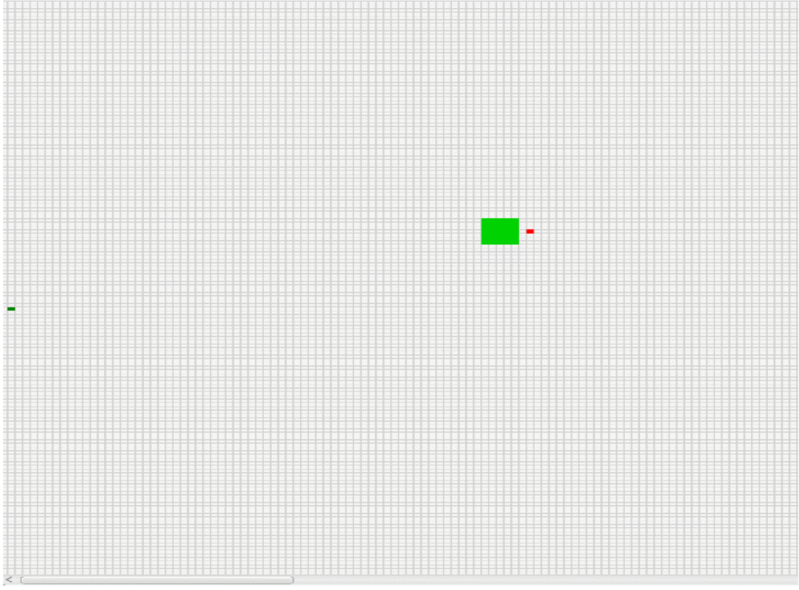


Figure 7.23 Test instance for the two exploration methods.

8. Further Research

A natural extension of JPS method would be to incorporate an additional rule or constraint, for finding non-dominated vertices based on radiation levels.

The model does not consider the size of the physical object that will be using the chosen path. The cells in the grid used in this model are 10×10 cm, a person would need more space and would not be able to fit between two obstacles with a gap of 10 cm. This consideration of object size might be added to the model as a constraint, or an object could be created to fit the path which has the size of an average human being. The speed and the objects possibilities when it comes to movement is also a subject for further research. For a human it is not natural to make a 90 degree turn. When for example, turning to the right we make a “curved” right turn.

This method is based on static and known instances. With a 3D gaming engine, like Unity or Unreal Engine, dynamic and unknown instances can be developed and used to test the method. Here we would not have full information over the map, and obstacles could move while using the path. This means that the path could now be blocked and not possible to use anymore. A new path would then be needed from a new starting point (where you got blocked) to the end point. An algorithm that can adapt to find feasible paths would be a helpful tool in these situations.

9. Conclusion

The scope of this thesis revolves around the path planning process in a radiation environment, as seen from the perspective of IFE's research dealing with radiation protection.

In this thesis, we described the development process of the software RadPathFinder. The software allows us to create a grid map with different types of radiation sources, different types of shielding and obstacles. In addition, it contains different path planning algorithms and provides a visualization tool to compare their performance.

The aim of the thesis was to provide decision support in the path planning process for nuclear plants. The problem background gave insight to why path planning is necessary when dealing with radiation. Due to the lack of real maps of nuclear plants, we created our own instances for evaluating the algorithms, and their assumptions and limitations were described. Two main conflicting goals were identified that had to be handled in this problem and such characteristic leads to a multi-objective path planning model to deal with this challenge, in which the objectives are:

- Minimize the total distance of the path
- Minimize the total accumulated dose rate

Regarding the research objective of finding an appropriate optimization method, the computational time was considered the most important aspect for comparison. A computational study with six different algorithms on nine different instances was performed. Three of the algorithms are applied to the shortest path problem without considering radiation, and the other three are modifications of these classical algorithms with radiation taken into account. The modified Dijkstra algorithm with radiation gave good results, but had some drawbacks, such as when the radiation is distributed irregularly and there are some areas where the radiation level is equal to zero, the algorithm does not have an instrument that can lead the search in areas without radiation, making the elapsed time increase drastically. The modified Jump Point Search method performed better than Dijkstra, even though there was an increase in computational time with a larger grid, it did not grow as much as with Dijkstra. The JPS method including radiation found paths found with a higher accumulated radiation dose, but in absolute numbers this is not significant. In terms of distance, the paths obtained with this method was better than those obtained with the radiation version of the Dijkstra algorithm.

The overall result from this study shows that the Jump Point Search with radiation method performs better when compared to the other tested algorithms on the same instances. Moreover, since the instances used for testing vary significantly in how the obstacles are placed in the map, we believe that this is a promising sign that the method proposed can work for different kind of room layouts.

References

Ahuja, R.K., Magnanti, T.L., Orlin, J.B. (1993). *Network Flows*. Prentice Hall. https://www.researchgate.net/publication/38009578_Network_Flows

Andersson, J., 2000. A survey of Multiobjective Optimization in Engineering Design. Technical Report: LiTH-IKP-R-1097. https://www.researchgate.net/publication/228584672_A_Survey_of_Multiobjective_Optimization_in_Engineering_Design

Bondy, A., & Murty, U.S.R., (2008). *Graph Theory*. Springer London. https://link.springer.com/book/9781846289699

Chatterjee, A., 2022. *Euclidean distance.* https://iq.opengenus.org/euclidean-distance/

Chatterjee, A., 2022. *Manhattan distance*. https://iq.opengenus.org/manhattan-distance/

Chen, C., Cai, J., Wang, Z., Chen, F., Yi, W., 2020. An improved A* algorithm for searching the minimum dose path in nuclear facilities. Prog. Nucl. Energy 126, 103394. https://www.sciencedirect.com/science/article/pii/S0149197020301463

Diaby, M., Karwan, M.H. (2016). Advances in Combinatorial Optimization: Linear Programming Formulations of the Traveling Salesman and Other Hard Combinatorial Optimization Problems. World Scientific. https://www.worldscientific.com/worldscibooks/10.1142/9725

Duchoň, F., Babinec, A., Kajan, M., Beňo, P., Florek, M., Fico, T., Jurišica, L., 2014. Path planning with modified A star algorithm for mobile robot. Procedia Engineering 96, 59 – 69. https://www.sciencedirect.com/science/article/pii/S187770581403149X

EPA., United States Environmental Protection Agency, 2022. *Protecting yourself from Radiation*. https://www.epa.gov/radiation/protecting-yourself-radiation

EPA., United States Environmental Protection Agency, 2022. *Radiation Health Effects*. https://www.epa.gov/radiation/radiation-health-effects

Han, J., & Koenig, S., 2021. A multiple surrounding point set approach using Theta* algorithm on eight-neighbor grid graphs. Info. Sciences 582, 618-632. https://www.sciencedirect.com/science/article/pii/S002002552101032X

Hilinski, J., 2022. *Using a grid to locate places*. Geogebra. https://www.geogebra.org/m/ny578g8v

Hofstad, K., 2019. Store norske leksikon. *Kjernekraft i Norge*. https://snl.no/kjernekraft_i_Norge

IFE, Institute for Energy Technology. 2022. *About IFE*. https://ife.no/en/about-ife/

IFE, Institute for Energy Technology. 2022. *Radiation Protection*. https://ife.no/en/division/radiation-protection/

Liu, Y.K., Li, M.K., Xie, C.L., Peng, M.J., Xie, F., 2014. Path-planning research in radioactive environment based on particle swarm algorithm. Prog. Nucl. Energy 74, 184–192. https://www.sciencedirect.com/science/article/pii/S0149197014000778

Liu, Y.K., Li, M.K., Xie, C.L., Peng, M.J., Wang, S.Y., Chao, M., Liu, Z.K., 2015. Minimum dose method for walking-path planning of nuclear facilities. Ann. Nucl. Energy 83, 161–171. https://www.sciencedirect.com/science/article/pii/S0306454915002121

Marler, T.R., & Arora, J.S., 2004. Survey of Multi-Objective Optimization Methods for Engineering. Struct Multidisc Optim 26, 369–395. https://www.researchgate.net/publication/225153273_Survey_of_Multi-Objective_Optimization_Methods_for_Engineering

Marler, T.R., & Arora, S.J., 2009. The weighted sum method for multi-objective optimization: new insights. Struct Multidisc Optim 41, 853–862. https://link.springer.com/content/pdf/10.1007/s00158-009-0460-7.pdf

Matai, R., Singh, S., Mittal, M.L. (2010). Traveling Salesman Problem: an Overview of Applications, Formulations, and Solution Approaches. https://www.intechopen.com/chapters/12736

Path Finding, n.d. Testing environment for different pathfinding heuristics. https://qiao.github.io/PathFinding.js/visual/

Patel, A. (2020) Heuristics. Amit's Thoughts on Pathfinding. http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html

Pei, Q.Y., Hao, L.J., Chen, C.H., Zheng, X.L., He. T., 2020. Minimum collective dose based optimal evacuation path-planning method under nuclear accidents. Ann. Nuclear Energy 147, 107644. https://www.sciencedirect.com/science/article/pii/S030645492030342X

Sapundzhi, F.I., & Popstoilov, M.S., 2017. Optimization algorithms for finding the shortest paths. Bulgarian Chemical Communications, Volume 50, Special Issue B, 115 – 120. http://bcc.bas.bg/BCC_Volumes/Volume_50_Special_B_2018/BCC-50-SI-B-Sapundzhi2-115-120.pdf

Statista, 2022. Primary energy consumption worldwide from 2000 to 2020. https://www.statista.com/statistics/265598/consumption-of-primary-energy-worldwide/#statisticContainer

Stewart, R.H., Palmer, T.S., DuPont, B., 2021. A survey of multi-objective optimization methods and their applications for nuclear scientist and engineers. Prog. Nucl. Energy 138, 103830. https://www.sciencedirect.com/science/article/pii/S0149197021001931

Szöke, I., Louka, M.N., Bryntesen, T.R., Bratteli, J., Edvardsen, S.T., RøEitrheim, K.K., Bodor, K., 2014. Real-time 3D radiation risk assessment supporting simulation of work in nuclear environments. J. Radiol. Prot. 34, 389–416. https://iopscience.iop.org/article/10.1088/0952-4746/34/2/389/meta

Weisstein, Eric W. *Graph Cartesian Product*. From Math World - A Wolfram Web Resource. https://mathworld.wolfram.com/GraphCartesianProduct.html

Weisstein, E.W. *Grid Graph*. From Math World – A Wolfram Web Resource. https://mathworld.wolfram.com/GridGraph.html

Wikipedia. 2022. *Shortest Path Problem*. https://en.wikipedia.org/wiki/Shortest_path_problem

Wikipedia. 2022. *Travelling Salesman Problem*. https://en.wikipedia.org/wiki/Travelling_salesman_problem

World Nuclear Association, October 2021. *Nuclear Power in the World Today*. https://world-nuclear.org/information-library/current-and-future-generation/nuclear-power-in-the-world-today.aspx

World Nuclear Association, November 2021. *World Energy Needs and Nuclear Power*. https://world-nuclear.org/information-library/current-and-future-generation/world-energy-needs-and-nuclear-power.aspx

Appendices

A-star Model

```
import com.example.radpathfinding.CanvasGridDisplay;
import javafx.scene.paint.Color;

import java.util.*;

public class AStarRad implements Algorithm {

private Instance graph;
private Map<Vertexes, Double> distance;
private Map<Vertexes, Double> radiation;
private Map<Vertexes, Double> fValueList;
private Map<Vertexes, Double> hValueList;
private Map<Vertexes, Double> radHValueList;
private Vertexes closestVertex;
private Vertexes startNode;
private Vertexes endNode;
private Set<Vertexes> endListOfNodes;
private CanvasGridDisplay cn;
private int xEndNode;
private int yEndNode;
private AStar.Heuristics heuristics;
private double radiationWeight;
private double distanceWeight;
private double accumulatedDose;
private String name;
private static final Double SPEED = 0.5;

public AStarRad(Instance graph, CanvasGridDisplay cn, AStar.Heuristics heuristics){
endListOfNodes = new HashSet<>();
hValueList = new HashMap<>();
distance = new HashMap<>();
fValueList = new HashMap<>();
radiation = new HashMap<>();
radHValueList = new HashMap<>();
this.graph = graph;
startNode = graph.getStartNode();
endNode = graph.getEndNode();
this.cn = cn;
xEndNode = endNode.getPositionX();
yEndNode = endNode.getPositionY();
this.heuristics = heuristics;
name = "A-Star";
}


public AStarRad(Instance graph, CanvasGridDisplay cn, AStar.Heuristics heuristics,
double radiationWeight, double distanceWeight,  int xSize, int ySize){
endListOfNodes = new HashSet<>();
hValueList = new HashMap<>();
distance = new HashMap<>();
fValueList = new HashMap<>();
```

```
radiation = new HashMap<>();
radHValueList = new HashMap<>();
this.graph = graph;
startNode = graph.getStartNode();
endNode = graph.getEndNode();
this.cn = cn;
xEndNode = endNode.getPositionX();
yEndNode = endNode.getPositionY();
this.heuristics = heuristics;
this.radiationWeight = radiationWeight;
this.distanceWeight = distanceWeight;
name = "A-star";


}


public void start(){
// we want the nodes with the lowest projected F value to be checked first

Queue<Vertexes> open = new PriorityQueue<>(Comparator.comparingDouble(a ->
fValueList.getOrDefault(a, 0d)));
open.add(startNode);
distance.put(startNode, 0.0);
radiation.put(startNode, 0.0);
computeH(startNode);
computeF(startNode);

while (!open.isEmpty()){
closestVertex = open.poll();
if (closestVertex == endNode){
break;
} else {
open.remove(closestVertex);
checkNeighbours(open);
endListOfNodes.add(closestVertex);
}
}
System.out.println("Dist A* rad:" + distance.get(endNode));
}

private void checkNeighbours(Queue<Vertexes> openNodeList){
for (Vertexes i: graph.getNeighbours(closestVertex)){
computeH(i);
if (endListOfNodes.contains(i) && radiation.get(i) + distance.get(i) >
radiation.get(closestVertex) +  getRadiation(i) * radiationWeight +
distance.get(closestVertex) + graph.getDistance(i, closestVertex) *  distanceWeight){
distance.replace(i, computeDistance(i) * distanceWeight);
radiation.replace(i, radiation.get(closestVertex) + getRadiation(i) * radiationWeight);
endListOfNodes.remove(i);
computeF(i);
openNodeList.add(i);
i.setParent(closestVertex);
cn.redrawCell(Color.OLIVE,i.getPositionX(),i.getPositionY());
}
else if (openNodeList.contains(i) && radiation.get(i) + distance.get(i) >
radiation.get(closestVertex) +  getRadiation(i) * radiationWeight +
distance.get(closestVertex) + graph.getDistance(i, closestVertex) *  distanceWeight){
distance.replace(i, computeDistance(i) * distanceWeight);
radiation.replace(i, radiation.get(closestVertex) + getRadiation(i) * radiationWeight);
computeF(i);
i.setParent(closestVertex);
cn.redrawCell(Color.GREEN, i.getPositionX(),i.getPositionY());
}
else if (!endListOfNodes.contains(i) && !openNodeList.contains(i)){
distance.put(i, computeDistance(i) * distanceWeight);
radiation.put(i, radiation.get(closestVertex) + getRadiation(i) * radiationWeight);
computeF(i);
openNodeList.add(i);
i.setParent(closestVertex);
i.setChecked();
cn.redrawCell(i.getPositionX(),i.getPositionY());
}
}
}

private void computeH(Vertexes currentNode){
int xCurrent = currentNode.getPositionX();
int yCurrent = currentNode.getPositionY();

switch (heuristics){
case Euclidean ->  hValueList.put(currentNode, (DistanceMeasurer.getEuclideanDist(
xCurrent, yCurrent, xEndNode, yEndNode, 0.1)));
case Manhattan ->  hValueList.put(currentNode, (DistanceMeasurer.getManhattan(
xCurrent, yCurrent, xEndNode, yEndNode, 0.1)));
}


}


private double computeDistance(Vertexes i){
return  distance.get(closestVertex) + graph.getDistance(i, closestVertex);

}


public double getAccumulatedRadiationLevel(){
Vertexes tempVertex = endNode;
accumulatedDose = 0;

int xPrev = 0;
int yPrev = 0;
double tempRad = 0;
while (tempVertex.getParent() != null){
xPrev = tempVertex.getPositionX(); yPrev = tempVertex.getPositionY();
tempRad = tempVertex.getRadiationLevel();
tempVertex = tempVertex.getParent();
tempRad = (tempRad + tempVertex.getRadiationLevel()) / 2;
accumulatedDose += (DistanceMeasurer.getEuclideanDist(xPrev, yPrev, tempVertex.getPositionX(),
tempVertex.getPositionY(), 0.1) / SPEED) * (tempRad / 3600);
}

System.out.println("Accumulated radiation level: " + accumulatedDose + " mSv");
System.out.println("Accumulated radiation level per year: "  +  accumulatedDose * 365);
return accumulatedDose;
}
```

```
private void computeF(Vertexes currentNode){
fValueList.put(currentNode, (radiation.get(currentNode) / 0.0043 ) * radiationWeight  +
( distance.get(currentNode) / 12)  * distanceWeight +
( hValueList.get(currentNode)  / 12 ) * distanceWeight);
}


private double getRadiation(Vertexes currentVertex){
int xCurrent = currentVertex.getPositionX();
int yCurrent = currentVertex.getPositionY();
switch (heuristics){
case Euclidean -> {
double dist = DistanceMeasurer.getEuclideanDist(
closestVertex.getPositionX(), closestVertex.getPositionY(), xCurrent, yCurrent, 0.1);
double time = dist / SPEED;
return time * currentVertex.getRadiationLevel() / 3600;
}
case Manhattan ->  {
double dist = DistanceMeasurer.getManhattan(
closestVertex.getPositionX(), closestVertex.getPositionY(), xCurrent, yCurrent, 0.1);
double time = dist / SPEED;
return time * currentVertex.getRadiationLevel() / 3600;
}
}
return  0;
}



public ArrayList<Vertexes> getSolutionNodes(){
try {
ArrayList<Vertexes> temp = new ArrayList<>();
Vertexes tempVertex = endNode;
temp.add(tempVertex);
while (tempVertex.getParent() != null){
tempVertex = tempVertex.getParent();
temp.add(tempVertex);
}
return temp;
} catch (NullPointerException d){
return null;
}

}

public double getPathLength(){
return distance.get(endNode);
}

public enum Heuristics{
Manhattan, Euclidean
}

@Override
public String getName(){
return name;
}

}
```

Dijkstra Model

```
public class DijkstraRadiation implements Algorithm{
    private Set<Vertexes> shortestPathTree;
    private Instance graph;
    private Map<Vertexes, Double> radition;
    private Map<Vertexes,Double> distance;

    private Vertexes closestVertex;
    private Vertexes startNode;
    private Vertexes endNode;
    private ArrayList <Vertexes> listOfAllNodes;
    private ArrayList <Vertexes> listOfEndVer;
    double max;
    private CanvasGridDisplay cn;
    private AStar.Heuristics heuristics;
    private String name = "Djikstra";
    private double accumulatedDose;
    private double speed = 0.5;

    public DijkstraRadiation(Instance graph, CanvasGridDisplay cn, AStar.Heuristics heuristics) {
        shortestPathTree = new HashSet<>();
        radition = new HashMap<>();
        distance = new HashMap<>();
        this.graph = graph;
        max = 0;
        startNode = graph.getStartNode();
        endNode = graph.getEndNode();
        listOfAllNodes = (ArrayList<Vertexes>) graph.getAllNodes();
        this.cn = cn;
        this.heuristics = heuristics;
        accumulatedDose = 0;
        listOfEndVer = new ArrayList<>();
    }
    public void start() {
        for (Vertexes n : listOfAllNodes) {
            radition.put(n, Double.MAX_VALUE);
        }
        Queue<Vertexes> open = new PriorityQueue<>((a, b) -> {
            // we want the nodes with the lowest projected F value to be checked first
            return Double.compare(radition.getOrDefault(a, 0d), radition.getOrDefault(b, 0d));
        });


        open.add(graph.getStartNode());

        radition.replace(graph.getStartNode(), 0.0);



        try{
            while (!open.isEmpty() && !shortestPathTree.contains(endNode)) {
                closestVertex = open.poll();
                shortestPathTree.add(closestVertex);
                closestVertex.setChecked();

               // cn.redrawCell(closestVertex.getPositionX(), closestVertex.getPositionY());
                updateDistances(open);
                open.remove(closestVertex);
            }
        }catch (NoSuchElementException we){
            System.err.println("Error");
        }
    }


    private void updateDistances(Queue<Vertexes> open) {
        for (Vertexes i : graph.getNeighbours(closestVertex)) {
            if (i.getMaterial() != Grid.Material.CHEKED && radition.get(closestVertex) + getRadiation(i) <
                    radition.get(i)) {
                radition.replace(i, getRadiation(i) + radition.get(closestVertex));
                i.setParent(closestVertex);
                open.add(i);
            }
        }
    }


    @Override
    public double getAccumulatedRadiationLevel(){
        Vertexes tempVertex = endNode;
        accumulatedDose = 0;
        int xPrev = 0;
        int yPrev = 0;
        double tempRad = 0;
        while (tempVertex.getParent() != null){
            xPrev = tempVertex.getPositionX(); yPrev = tempVertex.getPositionY();
            tempRad = tempVertex.getRadiationLevel();
            tempVertex = tempVertex.getParent();
            tempRad = (tempRad + tempVertex.getRadiationLevel()) / 2;
            accumulatedDose += (DistanceMeasurer.getEuclideanDist(xPrev, yPrev,
tempVertex.getPositionX(),
                    tempVertex.getPositionY(), 0.1) / speed) * (tempRad / 3600.0);
        }

        System.out.println("Accumulated radiation level: " + accumulatedDose + " mSv");
        System.out.println("Accumulated radiation level per year: "  +  accumulatedDose * 365);
        return accumulatedDose;
    }


    protected double getRadiation(Vertexes currentVertex){
        int xCurrent = currentVertex.getPositionX();
        int yCurrent = currentVertex.getPositionY();
        switch (heuristics){
            case Euclidean -> {
                double dist = DistanceMeasurer.getEuclideanDist(
                        closestVertex.getPositionX(), closestVertex.getPositionY(), xCurrent, yCurrent, 0.1);
                double time = dist / speed;
                return time * currentVertex.getRadiationLevel() / 3600;
            }
            case Manhattan ->  {
                double dist = DistanceMeasurer.getManhattan(
                        closestVertex.getPositionX(), closestVertex.getPositionY(), xCurrent, yCurrent, 0.1);
                double time = dist / speed;
                return time * currentVertex.getRadiationLevel() / 3600;
            }
        }
        return  0;
    }


    public void printSolution() {
        System.out.println("Distance: " + radition.get(endNode) + " m.");
    }

    @Override
    public double getPathLength(){
        return radition.get(endNode);
    }

    public ArrayList<Vertexes> getSolutionNodes(){
        ArrayList<Vertexes> temp = new ArrayList<>();
        Vertexes tempVertex = endNode;
        temp.add(tempVertex);
        while (tempVertex.getParent() != null){
            tempVertex = tempVertex.getParent();
            temp.add(tempVertex);
        }
        return temp;
    }


    @Override
    public String getName(){
        return name;
    }

}
```

JPS Model

```
import com.example.radpathfinding.CanvasGridDisplay;

import java.util.*;

public class JumpPointRad implements Algorithm{

    private Instance graph;
    private AStar.Heuristics heuristics;
    private Vertexes startNode;
    private Vertexes endNode;
    private CanvasGridDisplay cn;
    private int xEndNode;
    private int yEndNode;
    private final static String NAME = "JPS";

    private double SPEED = 0.5;

    public JumpPointRad(Instance graph, CanvasGridDisplay cn, AStar.Heuristics heuristics){
        this.graph = graph;
        startNode = graph.getStartNode();
        endNode = graph.getEndNode();
        this.cn = cn;
        xEndNode = endNode.getPositionX();
        yEndNode = endNode.getPositionY();
        this.heuristics = heuristics;
    }


    @Override
    public void start() {
        Map<Vertexes, Double> distance = new HashMap<>();
        Map<Vertexes, Double> fValueMap = new HashMap<>();
        Map<Vertexes, Double> hValueMap = new HashMap<>();

        Queue<Vertexes> openList = new PriorityQueue<>(Comparator.comparingDouble(a ->
fValueMap.getOrDefault(a, 0d)));

        Set<Vertexes> closedList = new HashSet<>();
        Map<Vertexes, Vertexes> patentMap = new HashMap<>();
        Set<Vertexes> goals = new HashSet<>();

        goals.addAll(graph.getNeighbours(endNode));
        goals.add(endNode);
        distance.put(startNode, 0.0);


        openList.add(startNode);
        while (!openList.isEmpty()){
            Vertexes currentNode = openList.poll();
            closedList.add(currentNode);
            if (currentNode == endNode){

                System.out.println("Ready");
                createPath();
                System.out.println("Distance: " + distance.get(endNode) + " m.");
                break;
            }
            findSuccessors(currentNode, distance, fValueMap, hValueMap, goals, openList, patentMap,
closedList);
        }
    }



    @Override
    public ArrayList<Vertexes> getSolutionNodes() {
        ArrayList<Vertexes> temp = new ArrayList<>();
        Vertexes tempVertex = endNode;
        temp.add(tempVertex);
        while (tempVertex.getParent() != null){
            tempVertex = tempVertex.getParent();
            temp.add(tempVertex);
        }
        return temp;
    }

    @Override
    public String getName() {
        return NAME;
    }

    @Override
    public double getPathLength() {
        return 0;
    }



    @Override
    public double getAccumulatedRadiationLevel() {
        return 0;
    }



    private void findSuccessors(Vertexes currentNode ,Map<Vertexes, Double> distance,
Map<Vertexes,Double> fValueList,
                                Map<Vertexes,Double> hValueMap, Set<Vertexes> goals, Queue<Vertexes> openList,
                                Map<Vertexes,Vertexes> parentMap, Set<Vertexes> closedList){

        Collection <Vertexes> neighbours = findNeighbours(currentNode);

        for (Vertexes neighbour: neighbours){

            Vertexes jumpNode = Jumping(neighbour, currentNode, goals);

            double heuristicDistance;
            double gDistance;

            if (jumpNode == null) {continue;}
            heuristicDistance = createPath(currentNode, jumpNode);
            gDistance = distance.getOrDefault(currentNode, 0d) + heuristicDistance;
            if (closedList.contains(jumpNode) && gDistance < distance.getOrDefault(jumpNode, 0d) ){
                distance.put(jumpNode, gDistance);
                fValueList.put(jumpNode, distance.getOrDefault(jumpNode, 0d) );
                jumpNode.setParent(currentNode);
                openList.add(jumpNode);
                closedList.remove(jumpNode);
            } else
                if (closedList.contains(jumpNode)){
                    continue;
                }


            if (!openList.contains(jumpNode) || gDistance < distance.getOrDefault(jumpNode, 0d)) {
                distance.put(jumpNode, gDistance);
                fValueList.put(jumpNode, distance.getOrDefault(jumpNode, 0d));
                jumpNode.setParent(currentNode);
                if (!openList.contains(jumpNode)) {
                    openList.offer(jumpNode);
                    openList.add(currentNode);
                    closedList.remove(currentNode);
                }
            }
        }
    }



    private double getDistance(Vertexes start, Vertexes goal){
        switch (heuristics){
            case Euclidean ->  {return DistanceMeasurer.getEuclideanDist(start.getPositionX(),
start.getPositionY(),
                    goal.getPositionX(), goal.getPositionY(), 0.1);}
            case Manhattan -> {return  DistanceMeasurer.getManhattan(start.getPositionX(),
start.getPositionY(),
                    goal.getPositionX(), goal.getPositionY(), 0.1);}
        }
        return 0d;
    }

    private Set<Vertexes> findNeighbours( Vertexes currentNode){

        Vertexes parent = currentNode.getParent();
        Set<Vertexes> neighbors = new HashSet<>();
        if (parent!= null){
            final int x = currentNode.getPositionX();
            final int y = currentNode.getPositionY();
            // get normalized direction of travel
            final int dx = (x - parent.getPositionX()) / Math.max(Math.abs(x - parent.getPositionX()), 1);
            final int dy = (y - parent.getPositionY()) / Math.max(Math.abs(y - parent.getPositionY()), 1);

            if (dx != 0 && dy != 0) {
                if (graph.isWalkable(x, y + dy))
                    neighbors.add(graph.getNode(x, y + dy));
                if (graph.isWalkable(x + dx, y))
                    neighbors.add(graph.getNode(x + dx, y));
                if (graph.isWalkable(x + dx, y + dy))
                    neighbors.add(graph.getNode(x + dx, y + dy));
                if (!graph.isWalkable(x - dx, y))
                    neighbors.add(graph.getNode(x - dx, y + dy));
                if (!graph.isWalkable(x, y - dy))
                    neighbors.add(graph.getNode(x + dx, y - dy));
            } else { // search horizontally/vertically
                if (dx == 0) {
                    if (graph.isWalkable(x, y + dy))
                        neighbors.add(graph.getNode(x, y + dy));
                    if (!graph.isWalkable(x + 1, y))
                        neighbors.add(graph.getNode(x + 1, y + dy));
                    if (!graph.isWalkable(x - 1, y))
                        neighbors.add(graph.getNode(x - 1, y + dy));
                } else {
                    if (graph.isWalkable(x + dx, y))
                        neighbors.add(graph.getNode(x + dx, y));
                    if (!graph.isWalkable(x, y + 1))
                        neighbors.add(graph.getNode(x + dx, y + 1));
                    if (!graph.isWalkable(x, y - 1))
                        neighbors.add(graph.getNode(x + dx, y - 1));
                }
            }
        } else {
            neighbors.addAll(graph.getNeighbours(currentNode));
        }

        return neighbors;
    }



    private Vertexes Jumping(Vertexes neighbor, Vertexes currentNode, Set<Vertexes> goals){
        if (neighbor == null || !graph.isWalkable(neighbor.getPositionX(), neighbor.getPositionY())) return
null;
        if (goals.contains(neighbor)) return neighbor;

        int dx = neighbor.getPositionX() - currentNode.getPositionX();
        int dy = neighbor.getPositionY() - currentNode.getPositionY();

        // check for forced neighbors
        // check along diagonal
        if (dx != 0 && dy != 0) {
            if ((graph.isWalkable(neighbor.getPositionX() - dx, neighbor.getPositionY() + dy) &&
                    !graph.isWalkable(neighbor.getPositionX() - dx, neighbor.getPositionY())) ||
                    (graph.isWalkable(neighbor.getPositionX() + dx, neighbor.getPositionY() - dy) &&
                            !graph.isWalkable(neighbor.getPositionX(), neighbor.getPositionY() - dy))) {
                return neighbor;
            }
            // when moving diagonally, must check for vertical/horizontal jump points
            if (Jumping(graph.getNode(neighbor.getPositionX() + dx, neighbor.getPositionY()), neighbor,
goals) != null ||
                    Jumping(graph.getNode(neighbor.getPositionX(), neighbor.getPositionY()+ dy), neighbor,
goals) != null) {
                return neighbor;
            }
        } else { // check horizontally/vertically
            if (dx != 0) {
                if ((graph.isWalkable(neighbor.getPositionX() + dx, neighbor.getPositionY() + 1)
&& !graph.isWalkable(neighbor.getPositionX(), neighbor.getPositionY() + 1)) ||
                        (graph.isWalkable(neighbor.getPositionX() + dx, neighbor.getPositionY() - 1)
&& !graph.isWalkable(neighbor.getPositionX(), neighbor.getPositionY() - 1))) {
                    return neighbor;
                }
            } else {
                if ((graph.isWalkable(neighbor.getPositionX() + 1, neighbor.getPositionY() + dy)
&& !graph.isWalkable(neighbor.getPositionX() + 1, neighbor.getPositionY())) ||
                        (graph.isWalkable(neighbor.getPositionX() - 1, neighbor.getPositionY() + dy)
&& !graph.isWalkable(neighbor.getPositionX() - 1, neighbor.getPositionY()))) {
                    return neighbor;
                }
            }
        }

        // jump diagonally towards our goal
        return Jumping(graph.getNode(neighbor.getPositionX() + dx, neighbor.getPositionY() + dy), neighbor,
goals);
    }


    private void createPath(){
        Vertexes tempNode = endNode;
        Vertexes parent;

        while (tempNode.getParent()!= null){
            parent = tempNode.getParent();
            double dx =  parent.getPositionX() - tempNode.getPositionX();
            double dy =  parent.getPositionY() - tempNode.getPositionY();


            if (dx != 0 && dy != 0){
                if (Math.abs(dx) > Math.abs(dy)){
                    double slope = dy / dx;
                    if (dx > 0){
                        double begPosY = tempNode.getPositionY();
                        for (int i = 0; i < Math.abs(dx); i++){
                            tempNode.setParent(graph.getNode(tempNode.getPositionX() + 1, (int)
Math.round(begPosY + slope)));
                            begPosY+=slope;
                            tempNode = tempNode.getParent();
                        }
                    } else
                    if (dx < 0){
                        double begPosY = tempNode.getPositionY();
                        for (int i = 0; i < Math.abs(dx); i++){
                            tempNode.setParent(graph.getNode(tempNode.getPositionX() - 1, (int)
Math.round(begPosY - slope)));
                            begPosY-= slope;
                            tempNode = tempNode.getParent();
                        }
                    }
                } else {
                    double slope = dx /dy;
                    if (dy > 0){
                        double begPosX = tempNode.getPositionX();
                        for (int i = 0; i < Math.abs(dy); i++){
                            tempNode.setParent(graph.getNode((int) Math.round(begPosX + slope),
tempNode.getPositionY() + 1));
                            begPosX+=slope;
                            tempNode = tempNode.getParent();
                        }
                    } else
                    if (dy < 0){
                        double begPosX = tempNode.getPositionX();
                        for (int i = 0; i < Math.abs(dy); i++){
                            tempNode.setParent(graph.getNode((int) Math.round(begPosX - slope),
tempNode.getPositionY() - 1));
                            begPosX-=slope;
                            tempNode = tempNode.getParent();
                        }
```

```
               }
            }
         } else
         if (dx == 0){
            if (dy > 0){
               for (int i = 0; i < Math.abs(dy); i++){
                  tempNode.setParent(graph.getNode(tempNode.getPositionX() , tempNode.getPositionY() +
1));
                  tempNode = tempNode.getParent();
               }
            }else {
               for (int i = 0; i < Math.abs(dy); i++){
                  tempNode.setParent(graph.getNode(tempNode.getPositionX(), tempNode.getPositionY() - 1));
                  tempNode = tempNode.getParent();
               }
            }
         } else {
            if (dx > 0){
               for (int i = 0; i < Math.abs(dx); i++){
                  tempNode.setParent(graph.getNode(tempNode.getPositionX() + 1,
tempNode.getPositionY()));
                  tempNode = tempNode.getParent();
               }
            }else {
               for (int i = 0; i < Math.abs(dx); i++){
                  tempNode.setParent(graph.getNode(tempNode.getPositionX() - 1, tempNode.getPositionY()));
                  tempNode = tempNode.getParent();
               }
            }
         }



      }
   }


   private double createPath(Vertexes currentNode , Vertexes endNode){

      double accumulated = 0;
      double dx = currentNode.getPositionX() - endNode.getPositionX();
      double dy = currentNode.getPositionY() - endNode.getPositionY();
      Vertexes prev = endNode;

      if (dx != 0 && dy != 0){
         if (Math.abs(dx) > Math.abs(dy)){
            double slope = dy / dx;
            if (dx > 0){
               double begPosY = endNode.getPositionY();
               for (int i = 0; i < Math.abs(dx); i++){
                  if (graph.isWalkable(currentNode.getPositionX() + 1, (int) Math.round(begPosY + slope))) {
                     accumulated += getRadiation(prev, graph.getNode(prev.getPositionX() + 1, (int)
Math.round(begPosY + slope)) );
                     prev = graph.getNode(prev.getPositionX() + 1, (int) Math.round(begPosY + slope));
                     begPosY += slope;
                  }
               }
            }
            else
            if (dx < 0){
               double begPosY = endNode.getPositionY();
               for (int i = 0; i < Math.abs(dx); i++){
                  if (graph.isWalkable(prev.getPositionX() - 1, (int) Math.round(begPosY - slope))){
                     accumulated += getRadiation(prev, graph.getNode(prev.getPositionX() - 1, (int)
Math.round(begPosY - slope)));
                     prev = graph.getNode(prev.getPositionX() - 1, (int) Math.round(begPosY - slope));
                     begPosY-= slope;
                  }
               }
            }
         } else {
            double slope = dx / dy;
            if (dy > 0){
               double begPosX = endNode.getPositionX();
               for (int i = 0; i < Math.abs(dy); i++){
                  if (graph.isWalkable((int) Math.round(begPosX + slope), prev.getPositionY() + 1)){
                     accumulated += getRadiation(prev, graph.getNode((int) Math.round(begPosX + slope),
prev.getPositionY() + 1));
                     prev = graph.getNode((int) Math.round(begPosX + slope), prev.getPositionY() + 1);
                     begPosX+=slope;
                  }
               }
            } else
            if (dy < 0){
               double begPosX = endNode.getPositionX();
               for (int i = 0; i < Math.abs(dy); i++){
                  if (graph.isWalkable((int) Math.round(begPosX - slope), prev.getPositionY() - 1)){
                     accumulated += getRadiation(prev, graph.getNode((int) Math.round(begPosX - slope),
prev.getPositionY() - 1));
                     prev = graph.getNode((int) Math.round(begPosX - slope), prev.getPositionY() - 1);
                     begPosX-=slope;
                  }
               }
            }
         }
      } else
      if (dx == 0){
         if (dy > 0){
            for (int i = 0; i < Math.abs(dy); i++){
               if (graph.isWalkable(prev.getPositionX() , prev.getPositionY() + 1)){
                  accumulated += getRadiation(prev, graph.getNode(prev.getPositionX() , prev.getPositionY()
+ 1));
                  prev = graph.getNode(prev.getPositionX() , prev.getPositionY() + 1);
               }
            }
         }else {
            for (int i = 0; i < Math.abs(dy); i++){
               if (graph.isWalkable(prev.getPositionX(), prev.getPositionY() - 1)){
                  accumulated += getRadiation(prev, graph.getNode(prev.getPositionX(), prev.getPositionY() -
1));
                  prev = graph.getNode(prev.getPositionX(), prev.getPositionY() - 1);
               }
            }
         }
      } else {
         if (dx > 0){
            for (int i = 0; i < Math.abs(dx); i++){
               if (graph.isWalkable(prev.getPositionX() + 1, prev.getPositionY())){
                  accumulated += getRadiation(prev, graph.getNode(prev.getPositionX() + 1,
prev.getPositionY()));
```

```
                        prev = graph.getNode(prev.getPositionX() + 1, prev.getPositionY());
                    }
                }
            }else {
                for (int i = 0; i < Math.abs(dx); i++){
                    if (graph.isWalkable(prev.getPositionX() - 1, prev.getPositionY())){
                        accumulated += getRadiation(prev, graph.getNode(prev.getPositionX() - 1,
prev.getPositionY()));
                        prev = graph.getNode(prev.getPositionX() - 1, prev.getPositionY());
                    }
                }
            }
        }


        return accumulated;


    }

    private double getRadiation(Vertexes source,Vertexes currentVertex){
        int xCurrent = currentVertex.getPositionX();
        int yCurrent = currentVertex.getPositionY();
        if (source == null) return 0.0;
        switch (heuristics){
            case Euclidean -> {
                double dist = DistanceMeasurer.getEuclideanDist(
                        source.getPositionX(), source.getPositionY(), xCurrent, yCurrent, 0.1);
                double time = dist / SPEED;
                return time * currentVertex.getRadiationLevel() / 3600;
            }
            case Manhattan ->  {
                double dist = DistanceMeasurer.getManhattan(
                        source.getPositionX(), source.getPositionY(), xCurrent, yCurrent, 0.1);
                double time = dist / SPEED;
                return time * currentVertex.getRadiationLevel() / 3600;
            }
        }
        return  0;
    }

}
```