Kinetic Travelling Salesman Problem: Multiple-Pursuer Defence of the Origin with Resupply

By Stefan Colakovic, Student #: 1621560

Supervised by Dr. Kathleen Steinhöfel

Originality Avowal:

I verify that I am the sole author of this report, except where explicitly stated to the contrary.

Signed:

Stefan Colakovic

X _____ .

Date: 23/04/2020

Abstract:

The contents of this report describe the design and implementation of an optimisation algorithm for a variant of the Kinetic Travelling Salesman Problem. A specific variant of this problem is being addressed: multiple-pursuer defence of the origin with resupply.

Defending the origin refers to preventing multiple moving targets from reaching the origin point of a graph. Pursuers must intercept all targets before they reach the origin, including making contact at the origin (i.e. waiting at 0,0 for a target).
Multiple-pursuer defence refers to the use of multiple pursuers, rather than one, to intercept the targets.
Resupply implies that, after each interception a pursuer makes, it must return to the origin before intercepting another target.

Optimisation of this problem refers to intercepting all targets in the minimal amount of time, and this report aims to detail a P-time method to generate an optimal allocation of pursuers to targets to satisfy this constraint. The paper is complemented by a C++ implementation of this method.

Acknowledgements:

I would like to thank Dr. Kathleen Steinhöfel for her support and attention during this project.

I would also like to thank Angelos Gkikas for his assistance and support on the project.

Contents:

Originality Avowal: 2
Abstract: 3
Acknowledgements: 3
Introduction: 5
Background: 6
Algorithm Design: 7
Software Design: 9
Implementation: 13
Evaluation: 24
Legal, Social, Ethical, and Professional Issues: 25
Conclusions and Further Work: 26
Bibliography: 27

Introduction:

The aim of this project is the design and implementation of an approximation algorithm for optimising a specific relaxation of the kinetic travelling salesman problem (henceforth referred to as KTSP).

KTSP itself is a generalisation of the travelling salesman problem, which is an NP-hard optimisation problem with many real-world applications; for example, the travelling salesman problem can be a model for many real-world logistics problems, or an abstract representation for decisions made in seemingly unrelated problems. It follows that certain relaxations of the KTSP may prove to be similarly useful to solving real-world problems, which provides value to investigating these relaxations. This is also simpler than investigating solutions to the general KTSP.

The specific relaxation being investigated is multiple-pursuer defence of the origin with resupply. This can be broken down into multiple separate relaxations simultaneously placed on KTSP: multiple pursuers, defence of the origin, and resupply.

Defence of the origin is a significant relaxation. It includes a Euclidean plane containing a set of targets. These targets move directly and linearly towards the origin (0, 0) at a certain fixed velocity. Pursuers begin at the origin and must intercept targets before they reach the origin. A notable part of this relaxation is that pursuers have a speed that is greater than or equal to all targets. Furthermore, meeting a target at the origin is also an acceptable interception.

Resupply is a relaxation that simplifies the decision-making of the problem but increases the time taken in the final solution. The essence of resupply is that pursuers must return to the origin after each interception they make. This means that it is no longer necessary to worry about a pursuer's position after an interception, which significantly reduces the state space of the problem.

Allowing multiple pursuers is an interesting relaxation, as it allows us to create significantly more optimal solutions while only slightly expanding the state space. Notably, problems tend to become trivial with a large enough population of pursuers, and it is common that strategies for multiple pursuers closely mimic strategies for single pursuers. This turns out to be especially true in the combination of all three relaxations above.

This creates a problem that is sufficiently relaxed to optimise in short time periods, while remaining relevant to real-world applications. For example, one direct application for the problem would be the creation of a missile defence system, choosing in which order to intercept dangerous targets to minimise time spent handling the situation. Another application would be the refuelling of incoming aircraft, allowing the optimal use of time and fuel for intercepting aircraft mid-flight and refuelling them before returning to airport. The problem seems to fit logistical problems very closely, and is thus a valuable research subject for business interests in that field.

Background:

There are three main sources that inspired this project and acted as the main research sources for the problem.

The first is a presentation by Angelos Gkikas given as part of a course on Optimisation Methods at King's College London
(Angelos Gkikas, 6CCS3OME & 7CCSMOME 18~19, "TSP and Similar Problems", https://keats.kcl.ac.uk/pluginfile.php/3874682/mod_resource/content/1/TSP_Introduction.pdf).
This presentation gave a description of the travelling salesman problem, showed its generalisation to the kinetic travelling salesman problem, and highlighted that there are relaxed variants of KTSP which may be more or less difficult to solve than the general KTSP. Specifically, 'Deadline Defence' as mentioned in the presentation is the more general form of the problem attempted in this paper.

The next source is an article published on the 'Approximation Results for Kinetic Variants of TSP' (Mikael Hammar, Bengt J. Nilsson, https://www.researchgate.net/publication/2357541_Approximation_Results_for_Kinetic_Variants_of_TSP).
This source was useful for conceptualising ways to approach KTSP problems, even if the variants solved were not directly relevant to this problem. Furthermore, certain concepts were applicable to the problem at hand, such as 'Lemma 1' from the paper stating 'An optimal salesman moves with maximal speed' (pg. 637, Mikael Hammar, Bengt J. Nilsson, https://www.researchgate.net/publication/2357541_Approximation_Results_for_Kinetic_Variants_of_TSP).

Finally, the paper 'The moving-target traveling salesman problem' (C.S. Helvig, Gabriel Robins, Alex Zlikovsky, 1998, https://www.cs.virginia.edu/~robins/papers/The_Moving_Target_Traveling_Salesman_Problem.pdf) had many significantly useful concepts for the problem approached in this report. Although the paper does not directly approach this specific relaxation, it approaches other similar combinations of relaxations; specifically, it covers 'Moving-Target TSP with Resupply after intercepts' (3., pg.162, 'The moving-target traveling salesman problem') and 'Defending the origin against incoming targets' (3.2., pg.166, 'The moving-target traveling salesman problem') with one pursuer, as well as a separate section on 'Multi-pursuer Moving-Target TSP with Resupply' (4., pg.168, 'The moving-target traveling salesman problem'). There were applicable theorems from each of these sections that could be combined to approach this specific problem.

Algorithm Design:

Initially, it must be explained why an approximation algorithm is being designed and implemented rather than a full optimisation algorithm. As proven in 'The moving-target traveling salesman problem' (C.S. Helvig, Gabriel Robins, Alex Zlikovsky, 1998, https://www.cs.virginia.edu/~robins/papers/The_Moving_Target_Traveling_Salesman_Problem.pdf), 'Theorem 14. Moving-Target TSP with Two Pursuers and Resupply is NP-hard when the objective is to minimize either the total time or the makespan, even when all targets have the same age' (pg.170). However, 'Theorem 11. For Moving-Target TSP with Resupply, when all targets move towards the origin, no valid tour is more than twice as long as an optimal valid tour' (pg.167) and 'Theorem 10. A tour which intercepts the targets in nondecreasing order of di/(−vi) is the slowest (i.e., worst) valid tour' (pg. 166). Based on these two theorems, we can create a valid tour, which is at worst double the length of the optimal tour, and incrementally improve it using a greedy algorithm to move closer to the optimal solution.

Our approximation algorithm will first find the danger of each target; the danger of a target refers to the amount of time it will take for the target to reach the origin, i.e. the target's distance to the origin divided by its speed. The targets will then be ranked in ascending order of danger. It can be noted that using one pursuer to intercept targets in this order is equivalent to the above mentioned 'Theorem 10'. As further mentioned in the above paper, 'A natural strategy would be to always intercept the least dangerous target unless intercepting that target would allow the most dangerous target to actually reach the origin' (pg. 167). The paper goes on to further detail that 'this simple strategy does not always yield an optimal tour' (pg. 167), but each application of this strategy does make an improvement over an allocation that only intercepts the most dangerous target. This shall be the strategy we use for our first pursuer, which may be denoted the 'critical pursuer' as it prevents our allocation from being invalid.

Additional pursuers will be used to further optimise this strategy. Since the critical pursuer ensures that our tour will be valid, we can freely send other pursuers directly for the least dangerous targets, improving our solution greatly with each such allocation. This strategy does have the capacity to reach the optimal solution – for example, when the number of targets is equal to the number of pursuers, this strategy reaches the natural solution of allocating a pursuer to each target.

This strategy can be implemented with the following steps:

Where there exist 'n' targets and 'p' pursuers,
While there exist targets that have not been intercepted:

> *Allocate pursuers 2-p to the least dangerous target if they are not already intercepting a target*
>
> *For pursuer 1 (the critical pursuer):*
> > *Iterate through targets from least dangerous to most dangerous.*
> > *If intercepting a target and returning to the origin takes less time than the danger of the most dangerous target:*
> >
> > *allocate pursuer 1 to that target, and*
> > *check whether pursuers 2-p are available to intercept targets, allocating them to the least dangerous if they are available*
> >
> > *Finally, allocate the most dangerous target to pursuer 1, except in the case that it has been allocated to another pursuer in the above step.*

Analysing this strategy shows that an approximation can be computed in O(n^2) (where p = 1 and we must iterate through all targets in each loop) and Ω(n) (where p ≥ n and we compute a full solution in the first step). Due to the use of pursuers 2-p, the problem has the capacity to be reduced by p - 1 at each loop, performing at Ω($\frac{n^2}{p-1}$) in these cases.

Comparing this strategy 's' to the optimal allocation 'o':

> In the best case where p ≥ n:
>
> s = o,
>
> by assigning the trivial solution of one pursuer per target.
>
> In the worst case where p = 1:
>
> s < 2o,
>
> by 'Theorem 11' of 'The moving-target traveling salesman problem' (pg. 167, C.S. Helvig, Gabriel Robins, Alex Zlikovsky, 1998, https://www.cs.virginia.edu/~robins/papers/The_Moving_Target_Traveling_Salesman_Problem.pdf) and an incremental improvement upon that worst tour (or the case that the worst tour is the only valid tour, in which case we reach s = o).
>
> Thus, in general,
>
> o <= s < 2o.

Software Design:

The primary focus of this project is the implementation of a system that approximates solutions to this problem. However, it is insufficient to simply implement the algorithm above. Instead, an implementation of this problem should create a modular structure for representing the problem with which further algorithms and solutions can be designed. This creates a general structure for analysis of this problem and will make further research simpler.

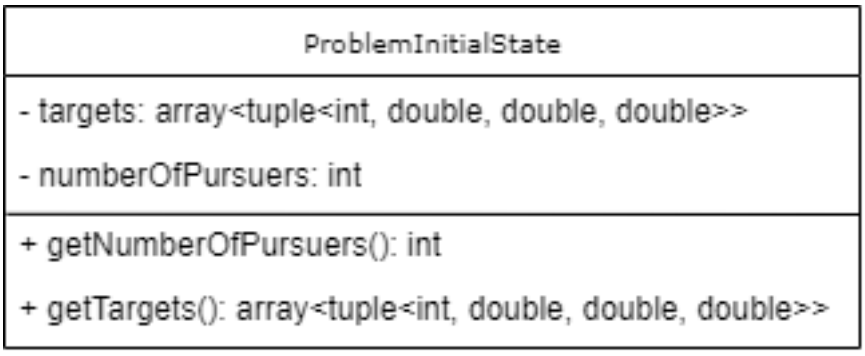
The software was designed with an agile philosophy (Mike Beedle et al, 2001, https://agilemanifesto.org/). In this case, this philosophy has been utilised by performing design and implementation in step with each other. This report will cover design and implementation in separate sections, but it is worthy of note that this does not necessarily reflect the chronological development of the software.

The most important question was how to represent a version of this problem on a computer. The first step to resolving this was to decide how to represent the targets. A natural representation takes from their starting properties as points on a Euclidean plane; the tuple of coordinates (x, y) can be extended to the tuple (ID, x, y, speed). This contains all the relevant information about a target; the velocity can be represented as speed since it will always be in the direction of the origin, and the distance to the origin can be computed via trigonometry using the Pythagorean theorem (Pythagoras, ~550BCE). The 'ID' will be an integer enumeration along all targets, used to quickly identify which target is being referred to. These tuples can be stored in an array, creating a sort of two-dimensional array of information.

In addition to representing targets, we must represent the pursuers. This is significantly easier: part of the relaxation of the problem is that pursuers all start at the origin and move with speed '1'. Furthermore, they can be implicitly identified by their location in an array, replacing their need for an ID. Thus, the only information we need about pursuers is an integer denoting how many pursuers exist.

Finally, the class' constructors and destructors trivially assign and destroy the fields mentioned.

This can be compounded into a 'Problem Initial State' class which contains the above information. This class can be represented by the following UML diagram:

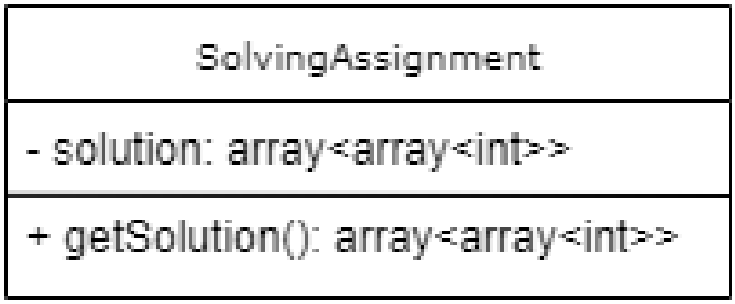


Next, we should define how a solution to a problem will be stored. Based on the ProblemInitialState class, we can create a solution with a 2D array of integers, where a column refers to a pursuer's interception targets in order. For example:

[1][2]
[3][7]-------->This vector would imply pursuer 1 (the first column) goes for target 1, then 3,
[6][5] then 6, then 4, while pursuer 2 (the second column) would
[4] go for target 2, then 7, then 5.

This makes use of the IDs declared in ProblemInitialState. This is the only information we need to declare a solution, as all other information about the problem can be inferred by combining a solution with a problem's initial state.

This can be compounded into a 'Solving Assignment' class which can be represented by the following UML diagram:



To make use of these, we will need to create a class which solves a given ProblemInitialState and generates a SolvingAssignment based on it. In order to support future iterations of the program, it would be wise to create an interface which contains the signatures used for this purpose and have actual implementations of specific algorithms be subclasses which override the methods of this interface. This will allow a general outside class to generically reference any assignment algorithm through this interface, rather than being explicitly tied to any one assignment algorithm.

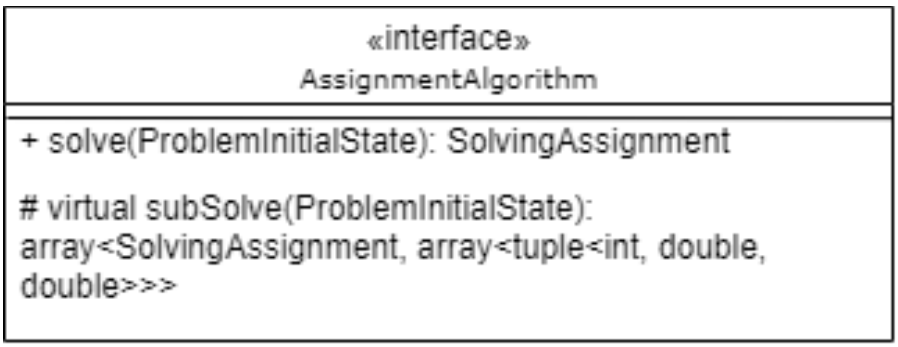
This 'Assignment Algorithm' interface will consist of two methods: 'solve' and 'subSolve'.

'subSolve' will be the method where the solution is created. It will be a protected virtual method, meaning that there is no base implementation for it in the interface, but children of some subclass can use their parents' 'subSolve' methods, allowing for simple extensions to existing assignment algorithms. The key component of 'subSolve' is that it returns both a finished SolvingAssginment as well as the ordered danger ranking computed within the subSolve method, meaning that the same table does not need to be computed twice. This is important since computing a sorted table via comparison is on the order of O(n log n) at best, and thus avoiding repetition saves considerable time.

'solve' will be a public method that is implemented directly in the AssignmentAlgorithm interface. The only purpose of 'solve' will be running 'subSolve' within it, properly deleting the ordered target ranking table, and passing the finished SolvingAssignment. This allows an external class to call 'solve' from any AssignmentAlgorithm and consistently receive a SolvingAssignment alone, without being tightly coupled to the inner function of any solving algorithm.

There were considerations for making 'solve' and 'subSolve' static methods, to prevent the need for instantiating any objects and simply referencing the classes. However, the benefits of this approach are minimal, as only slight overhead memory would be saved. It was decided that the increased complexity was not worth the minor benefits in the end.

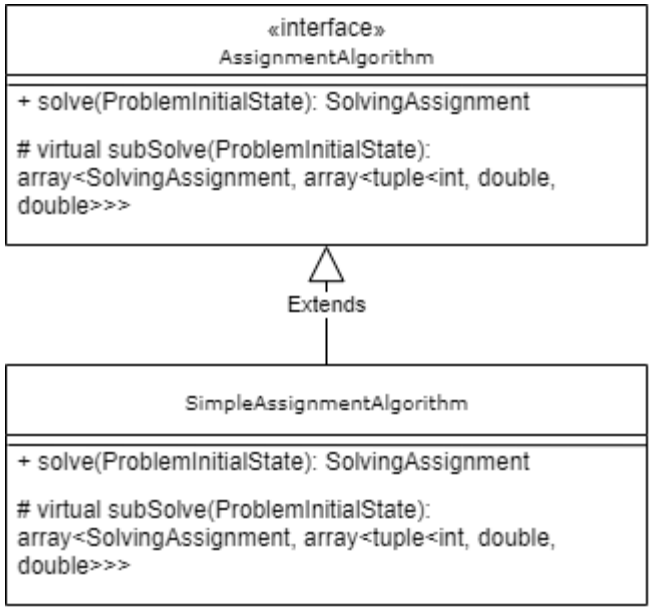
This can be compounded into a 'Assignment Algorithm' class which can be represented by the following UML diagram:



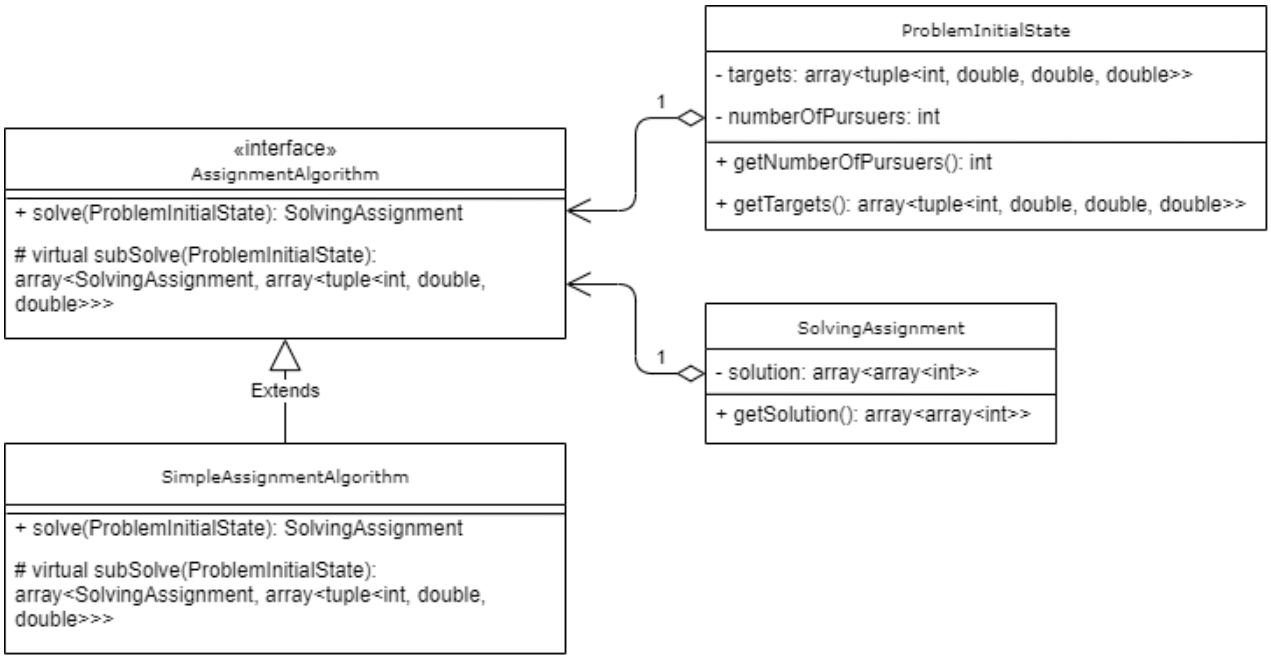
Utilising the above interface, we will create a subclass to implement a simple assignment algorithm as an example. We can use the algorithm created in 'Algorithm Design' for this.

Creating a subclass of AssignmentAlgorithm, we implement the previously virtual 'subSolve' method by following the procedure outlined in the earlier chapter. This creates a complete class which can be instantiated, as opposed to an interface.

This relationship can be shown in UML with the following diagram:



Overall, the design space of the project can be represented in UML as follows:



Implementation:

For the implementation of the software, I chose to use C++ (Bjarne Stroustrup, 1985). C++'s support for object-oriented programming, as well as standard libraries such as 'vector', 'tuple', and 'find', will greatly support the creation of the software designed above.

As a compiler and platform for C++, I chose to use Unreal Engine 4 (Epic Games, Inc., www.unrealengine.com/). This is to facilitate many useful extensions for future work to the project. Unreal Engine 4's capacity for a graphical interface presents an avenue for many useful features, such as visualisations, interactive problem generators, and demonstrating real-world analogues of the problem. Furthermore, the inclusion of Unreal Engine 4 into the files above is minimal and would be trivial to remove; this allows us to retain the modularity of our solution while effectively extending its use.

The first component implemented was the ProblemInitialState class. This follows the schematic of the design portion. As is common with C++ classes, the class is split into two files: a header file named 'ProblemInitialState.h', and a body file named 'ProblemInitialState.cpp'. The purpose of this is to allow outside files to see the signatures of the ProblemInitialState class, through the header file, while keeping the behaviour of the class obscured within the body file.

The major change is the use of the 'std::vector' standard library class in place of the generic 'array' class named in the design. This provides some useful methods, such as iterators and common functions (e.g. 'push' and 'pop' as they would function on a stack), which serve to make later implementation more efficient and less complex. The vector object can also be used in all the same ways as an array, and the increase in overhead is very minor.

It is notable that, for all classes in the design, 'vector' has been used to replace 'array'. Below is an example of 'vector' used as such in ProblemInitialState.h:

```
class KTSP_MULTIPLE_DEFEND_API ProblemInitialState
{
public:
	.
	.
	.
private:
	vector<tuple<int, double, double, double>> targets;
		//Vector of targets: tuple - ID, x position, y position, speed (<= 1)
	.
	.
	.
};
```

The constructor for ProblemInitialState simply takes arguments equivalent to the fields of the class and assigns them to their respective fields:

```
ProblemInitialState::ProblemInitialState(vector<tuple<int, double, double, double>>
targets, int numberOfPursuers)
	:targets(targets), numberOfPursuers(numberOfPursuers)
{
}
```

A secondary constructor was considered for reading input from a file but was not completed. It remains as a comment in the code. This would be a possible extension to the project.

The default destructor is used, as the fields stored are direct types and not pointers.

There exist two getter methods that return the fields of the class. These are fairly standard, with one notable exception:

```
vector<tuple<int, double, double, double>>* ProblemInitialState::getTargets()
{
	return &targets;
}
```

This getter method returns a pointer to the 'targets' field rather than the field itself. This is intended to allow only a pass by reference, preventing the program from unintentionally containing two copies of the same data in memory. It is notable that this is not performed for the other getter, as passing a pointer does not save any memory over passing an int.

Next, the SolvingAssignment class was created. This is a very simple class, containing only a field for the solution and a constructor for initialising said field:

```
class KTSP_MULTIPLE_DEFEND_API SolvingAssignment
{
public:
	SolvingAssignment(vector<vector<int>>* solution);
	~SolvingAssignment();

private:
	vector<vector<int>>* solution; …
};
```

Again, it is notable that the 'solution' field is a pointer rather than a direct type. In this case, the field itself is a pointer, not only a method referring to 'solution'. This is because the 'solution' vector will usually be created in an external class before the SolvingAssignment object is instantiated. This allows us to pass the existing vector to the SolvingAssignment object, rather than creating a copy of it, saving memory.

Arguably the most important part of this project is the implementation of the 'AssignmentAlgorithm' interface. In C++, interfaces are treated as 'virtual classes' which cannot be instantiated. This is very useful, as it allows us to partially implement the class. This partial implementation will be inherited by all subclasses, and the subclasses then only need to implement the virtual methods to gain full functionality.

Firstly, we can look at the virtual part of the class:

```
class KTSP_MULTIPLE_DEFEND_API AssignmentAlgorithm
{
public:
	…
protected:
	virtual tuple<SolvingAssignment*, vector<tuple<int, double, double>>*>*
	subSolve(ProblemInitialState* problem) = 0;
		//This should be where the majority of work is done for the algorithm
};
```

This method provides a signature for children to fulfil when overriding this method. This signature can be broken down into two important parts within a tuple:

'`SolvingAssignment*`' is a solution to the problem as implemented by the class. This can be a partial solution or a final one. The reason why we do not only return this is elucidated by the other part of the tuple, '`vector<tuple<int, double, double>>*`'. This second part will be a useful partial calculation within a solution: the sorted array of targets by their danger. The values in this are very similar to within the 'targets' array of a ProblemInitialState, except instead of a set of 'x, y' values in the tuple, we store only the danger value of that target. This makes it significantly more efficient to do further calculations on danger, rather than having to re-compute it at each step that needs it. If we wanted to create a secondary subclass (effectively a 'grandchild' of the original AssignmentAlgorithm interface), the grandchild might want to make use of some of the working from the original child. By passing the sorted danger array, we prevent needing to compute it twice. This specific array was chosen to be passed along since it is a significantly expensive array to compute; using comparison-based sorting has a minimum of O(n log n) ('Introduction to Algorithms', Thomas H. Cormen et al, 1990). Repeating this kind of sorting would be very wasteful, and so we pass this array by pointer to preserve both computation time and memory.

At the very end, however, we will not need the sorted danger array. In order to ease the burden for future iterations of the software, AssignmentAlgorithm has a non-virtual method 'solve' which properly deletes and discards the danger array, passing only the SolvingAssignment pointer:

```
SolvingAssignment* AssignmentAlgorithm::solve(ProblemInitialState* problem) //Moved
this to be a generic method in AssignmentAlgorithm
{
	tuple<SolvingAssignment*, vector<tuple<int, double, double>>*>* result = this-
>subSolve(problem);

	SolvingAssignment* solution;
	vector<tuple<int, double, double>>* targetDangerRankedList;
	std::tie(solution, targetDangerRankedList) = *result;

	delete targetDangerRankedList;
	delete result;
	return solution;
}
```

This requires careful management of memory due to the use of pointers. 'subSolve' passes a pointer to a tuple of pointers. We separate these pointers into three distinct variables. We then delete the pointer to the array containing the sorted danger ranked list; this removes the array itself from memory. We then delete the pointer to the tuple we received from 'subSolve'; this removes the tuple from memory, but notably, the 'SolvingAssignment' pointed to by the first element of the tuple remains in memory. This allows us to return a pointer to the SolvingAssignment generated.

Implementing the above in a base class means that programmers implementing subclasses do not need to worry as much about memory management of their returned values, and simply need to return pointers. This makes maintenance and extension of the code easier and neater.

Finally, as a sample for our constructed framework, the 'SimpleAssignmentAlgorithm' class will be implemented. This class shall be a child of AssignmentAlgorithm, and override the virtual subSolve method:

```
class KTSP_MULTIPLE_DEFEND_API SimpleAssignmentAlgorithm : public AssignmentAlgorithm
{
protected:
	tuple<SolvingAssignment*, vector<tuple<int, double, double>>*>*
	subSolve(ProblemInitialState* problem);
	…
};
```

The rest of the header contains only the default constructor and destructor for the class.

The contents of SimpleAssignmentAlgorithm.cpp detail the implementation of our earlier designed algorithm for an approximation of the problem within the subSolve method. There are a few interesting details that are specific to the implementation as opposed to the originally designed pseudocode procedure, and so the following section will highlight parts of the code.

```
	//Instantiate solution vector (see SolvingAssignment.h for definition)
	vector<vector<int>> solution = {};

	//Add 2D vectors for each pursuer
	for (int i = 0; i < problem->getNumberOfPursuers(); i++)
	{
		solution.push_back({});
	}
```

Here, we utilise the 'push_back(…)' method of the vector object. This makes it significantly easier to add objects to vectors, as opposed to arrays, as we do not need to manually track how many elements are in the vector.

Next in the code…

```
	//Create ranked list of targets by danger
	vector<tuple<int, double, double>> targetDangerRankedList = {};

	//Break if there are no targets; no activity is the correct activity for the
	empty problem
	if (targets->empty())
	{
		return packageTheSolution(solution, targetDangerRankedList);
	}
```

This is a simple check to end the algorithm early if there are no targets. This is simple validation, but gives a chance to explain a subroutine declared earlier in the code, ‘packageTheSolution’:

```
tuple<SolvingAssignment*, vector<tuple<int, double, double>>*>*
packageTheSolution(
vector<vector<int>> &solution,
vector<tuple<int, double, double>> &targetDangerRankedList)
{
	SolvingAssignment* solved = new SolvingAssignment(&solution);
	return new tuple<SolvingAssignment*, vector<tuple<int, double, double>>*>
(std::make_tuple(solved, &targetDangerRankedList));
}
```

This is a subroutine that takes references to ‘solution’ (as is required to instantiate a SolvingAssignment) and ‘targetDangerRankedList’ (the sorted array of danger values), puts ‘solution’ into a new SolvingAssignment, and then returns these variables in the signature required by the subSolve method. It is created as a subroutine to prevent code repetition; rather than re-iterating this process each time we must return a value, we simply return what is returned by this function. It will also allow us to edit all instances where we return, making it simpler to refactor the code.

Returning to 'subSolve'…

```
for (int i = 0; i < targets->size(); i++) //For all targets in the problem...
{
	tuple<int, double, double, double> target = (*targets)[i];
	int targetID = get<0>(target);
	double targetDanger = sqrt(pow(get<1>(target), 2) + pow(get<2>(target), 2)) /
get<3>(target);

	tuple<int, double, double> targetTuple = std::make_tuple(targetID,
	targetDanger, get<3>(target));	//Make a tuple out of the target's ID, it's
	calculated danger value, and its speed

	auto iter = targetDangerRankedList.begin();	//Insert using linear search;
	currently O(n^2), could be O(n*log2(n)) if Binary Search used, consider using
	std::upper_bound for this

	while (targetComparison(targetTuple, *iter) == false)
	{
		iter++;
	}

	targetDangerRankedList.insert(iter, targetTuple);	//Insert it before that
	found tuple - i.e. sort them in order of ascending danger value
}
```

Our next section is taking the list of targets, as in the ProblemInitialState, and sorting them into ascending order of danger. We do this by an insertion sort ('Algorithms', Robert Sedgewick, 1983), as the simplest implementation. This would take O(n^2) time to create a sorted list, and is a point where future iterations of the algorithm could be improved. For example, by using binary search rather than linear search, this could be reduced to O(n log n). However, as shall be demonstrated later, this does not impact the general worst-case running time of our algorithm, and so is not the most pressing improvement.

Another notable part of this segment:

```
sqrt(pow(get<1>(target), 2) + pow(get<2>(target), 2)) / get<3>(target);
```

This calculates a target's danger by application of Pythagoras' theorem to find the distance to the origin, then divides that by the target's speed to find the danger.

Finally, this section of code uses another subroutine declared earlier in the code, 'targetComparison':

```
bool targetComparison(tuple<int, double, double> &leftTarget, tuple<int, double,
double> &rightTarget)
{
	if (get<1>(leftTarget) < get<1>(rightTarget))
	{
		return true;
	}
	else
	{
		return false;
	}
}
```

This is simply a Boolean function to determine if one target should be sorted before another, returning true if it should lie to the left and false otherwise. This is used to sort the list into ascending order. It is defined as a subroutine as it facilitates changing the sorting routine to binary search in further work on the topic.

Finally, we reach the part of subSolve where our algorithm begins assigning pursuers to targets. To do this, we establish a set of variables on our existing data:

```
//Important info
auto mostDangerousUncaught = targetDangerRankedList.begin();
double timePassed = 0.0; //This will be adding lots of small floats together into a
sum; consider using Kahan Summation Algorithm to avoid error

vector<bool> targetsCaught(targetDangerRankedList.size(), false);
auto mostDangerousBool = targetsCaught.begin();
vector<double> busyUntil(problem->getNumberOfPursuers() - 1, 0.0);
```

This most prominently includes an array of Boolean values representing whether a target at position ‘n’ within the target danger array has been intercepted already. This array is used to know when we have intercepted all targets.

Furthermore, we introduce a ‘busyUntil’ array, which denotes how long pursuers 2-p are away from the origin. We use this in conjunction with the ‘timePassed’ variable to know whether we are free to assign more targets to the non-critical pursuers. It is important to note that ‘timePassed’ is a calculated value, and is not based on real-time.

While mentioning ‘timePassed’, it will be valuable to mention an imprecision induced by this implementation: the use of ‘double’. The type used to calculate many of the numerical values in this implementation is ‘double’, which is a 64-bit floating point number. This introduces inaccuracies based on floating point arithmetic, such as truncation and overflow. Furthermore, it only contains values up to a certain precision, meaning that some details may be lost. This issue is difficult to resolve, as it requires the implementation of a full computational system for representing real numbers without loss of precision, which is not native to C++. This is a possible extension to the project in future iterations.

Furthermore, one specific error will be most prevalent in this solution. Since ‘timePassed’ will be calculated using multiple floating point additions, the error of each addition will be compounded upon this variable. The use of the Kahan summation algorithm (‘Further remarks on reducing truncation errors’, William Kahan, 1965) would compensate for this issue, and would be a valuable addition to the program in further work.

These variables are now used to create a loop equivalent to the procedure defined in the 'Algorithm Design' chapter:

```
while (std::find(targetsCaught.begin(), targetsCaught.end(), false) !=
targetsCaught.end())
{
	//Time until most dangerous target hits the origin
	double greatestDanger_timeUntilCollission = get<1>(*mostDangerousUncaught) -
		timePassed;

	//Assign all the available non-critical pursuers to the least dangerous targets
	if (assignNonCriticalPursuers(timePassed, targetsCaught, busyUntil,
	targetDangerRankedList, solution))
	{
		break;
	}

	//Pursuer 1: go through targets, from least to most dangerous, and grab those
	that you can without missing the most dangerous target, then finally grab the
	most dangerous
	auto dangerIter = --targetDangerRankedList.end();
	auto caughtIter = --targetsCaught.end();
	while (dangerIter > mostDangerousUncaught)
	{
		if (findRoundtripTime(*dangerIter, timePassed) <=
		greatestDanger_timeUntilCollission && !(*caughtIter))
		{
			timePassed += findRoundtripTime(*dangerIter, timePassed);
			greatestDanger_timeUntilCollission -= timePassed;
			*caughtIter = true;
			solution[0].push_back(get<0>(*dangerIter));

			//When a target gets intercepted and time passes, it is possible
			non-critical pursuers are free; try and use these after each
			Pursuer1 assignment to remain optimal
			if (assignNonCriticalPursuers(timePassed, targetsCaught,
			busyUntil, targetDangerRankedList, solution))
			{
				break;
			}
		}
		dangerIter--;
		caughtIter--;
	}

	if (!*mostDangerousBool)
	{
		timePassed += findRoundtripTime(*mostDangerousUncaught, timePassed);
		*mostDangerousBool = true;
		solution[0].push_back(get<0>(*mostDangerousUncaught));
		mostDangerousBool++;
		mostDangerousUncaught++;
	}
}
```

The comments of this section describe well how its components match the designed algorithm's performance. Some values are duplicated in memory, such as 'greatestDanger_timeUntilCollission', in order to reduce the computational steps required.

Two important subroutines are used within this structure, ‘findRoundtripTime’ and ‘assignNonCriticalPursuers’:

```
double findRoundtripTime(tuple<int, double, double> &target, double &timePassed)
{
	return (((get<1>(target) - timePassed) * get<2>(target)) / (1 +
	get<2>(target))) * 2;
}
```

This is a procedure used to find the time it would take for a pursuer and target to meet. It is calculated by finding the distance between the target and the origin, and using the combined speed of the target and pursuer to calculate how long it would take for them to meet. This value is then doubled to account for the pursuer needing to return to the origin to refuel. It is notable that ‘timePassed’ is included in this calculation – it is used in its factorised form to account for how far the target has already travelled by the time the pursuer begins its route to intercept it.

```
bool assignNonCriticalPursuers(double &timePassed, vector<bool> &targetsCaught,
vector<double> &busyUntil, vector<tuple<int, double, double>> &targetDangerRankedList,
vector<vector<int>> &solution)
{
	auto dangerIter = --targetDangerRankedList.end();
	auto caughtIter = --targetsCaught.end();
	bool jobsDone = false;

	for (int i = 1; i <= busyUntil.size() && !jobsDone; i++)	//For each non-
	critical pursuer; 'i' is set up to properly reference the pursuers in solution
	{
		if (busyUntil[(i - 1)] <= timePassed) //If the 'i'th pursuer is at the
		origin and not chasing a target...
		{
			//Iterate backwards through targets until reaching either an
			unassigned target OR the beginning of all targets
			while (*caughtIter)
			{
				if (caughtIter == targetsCaught.begin())
				{
					jobsDone = true;
					break;
				}
				else
				{
					dangerIter--;
					caughtIter--;
				}
			}

			//Either all targets have been reached, or we can assign pursuer
			i to the target at DangerIter
			if (!jobsDone)
			{
				busyUntil[(i - 1)] += findRoundtripTime
							(*dangerIter, timePassed);
				solution[i].push_back(get<0>(*dangerIter));
			}
		}
	}

	//Return the bool showing whether we finished the solution
	return jobsDone;
}
```

This subroutine is used to allocate targets to each non-critical pursuer. It is used in multiple locations and is thus kept in a subroutine to minimise code repetition. It is also designed to return a Boolean value denoting whether every target has been intercepted, which is used in the main loop to break out of the while loop early and save processor time from being wasted once the problem is solved. This must be done separately to the critical pursuer, as it has a different process to allocate its targets.

It is worth noting that, due to nested loops, this seems at first glance to run at O(n^4). However, it should be noted that the problem is reduced in size each time a non-critical pursuer is allocated a target, effectively maintaining the O(n^2) complexity as the pseudocode procedure.

Evaluation:

The solution implemented above is a P-time approximation algorithm for an NP problem, with upper-bound performance of double the optimal solution, which is accomplishes the goal of the project specification. The implementation itself is modular and maintainable, allowing for future work to be easily implemented. Future work is also valuable in the subject – for example, comparison of the problem to NP-hard problems – meaning that maintainability was an important factor for the project. This was accomplished well in the final product.

A major drawback of the implementation is the use of double floating-point numbers. This introduces inaccuracy and error. The inaccuracy should not introduce erroneous behaviour within the code but may provide inaccurate allocations in some cases. Furthermore, some parts of the code are not fully optimal, such as the sorting of targets by danger and the summation of time passed. These overall detract from the quality of the final solution.

The theoretical aspects of the project could use further research; an approximation algorithm has been designed, but an algorithm which fully optimises the problem would also be a valuable area for research. Furthermore, during research, certain opportunities for further research became apparent; for example, the effects of disallowing stationary targets on the problem's complexity, and whether it may reduce the problem from NP to P.

Overall, the results produced were satisfactory and produced to a high quality. The drawbacks could be remedied with future work, and further theoretical value could be gained both from utilising the implementation as well as building upon the algorithmic design within this report.

Legal, Social, Ethical, and Professional Issues:

Legal:

Unreal Engine 4 is licensed software, and royalties must be paid if it is used for commercial purposes. Due to the implementation's educational nature, the license does not require royalties be paid on the current implementation. However, if further work was done and eventually commercialised, this is a legal boundary that must be kept in mind.

Professional:

The legal aspects would be a significant boundary in professional commercial work using the project. It would decrease income generated from the solution. Professional educational work on the subject does not suffer from this issue, and so further research on the topic does not suffer the above professional issue.

Social:

Public wellbeing is not realistically impacted by this project, and environmental concerns are not created by it. Certain real-world applications might raise these issues; for example, public wellbeing might be affected if the research is used in a military solution. However, these aspects are far removed from the contents of the project, and would require significant external work – this constitutes '*novus actus interveniens'*, breaking the chain of causation and clearing the report of liability.

Ethical:

The project follows the British Computer Society's Code of Conduct. The research and implementation are accessible, beneficial to the public interest, constructed with professional competence, respectful to their relevant authority, and respectful to the profession. Few ethical issues could arise from the project as is contained in this report.

Conclusions and Further Work:

The implementation of this project serves as an excellent starting point for further research in the subject. Investigation of the relation between P and NP problems is a significant field within Computer Science, and the implementation granted provides a platform for statistical experimental evaluation of an unexplored variant of an NP-Hard problem. Furthermore, the implementation can be easily extended to cover more variants of the problem.

Further work specifically applying to the implementation would include the implementation of the Kahan summation algorithm, the switch to binary sorting rather than linear sorting, and the replacement of the double floating-point type with a type that does not lose precision. These changes would improve the quality of the final solution without impacting its purpose or function. Furthermore, adding an implementation of a complete optimisation algorithm (as opposed to the given approximation algorithm) would allow for comparison between the performance of both cases.

Bibliography:

Optimisation Methods at King's College London
(Angelos Gkikas, 6CCS3OME & 7CCSMOME 18~19, "TSP and Similar Problems",
https://keats.kcl.ac.uk/pluginfile.php/3874682/mod_resource/content/1/TSP_Introduction.pdf)

Approximation Results for Kinetic Variants of TSP
(Mikael Hammar, Bengt J. Nilsson,
https://www.researchgate.net/publication/2357541_Approximation_Results_for_Kinetic_Variants_of_TSP).

Approximation Results for Kinetic Variants of TSP
(pg. 637, Mikael Hammar, Bengt J. Nilsson,
https://www.researchgate.net/publication/2357541_Approximation_Results_for_Kinetic_Variants_of_TSP)

The moving-target traveling salesman problem
(C.S. Helvig, Gabriel Robins, Alex Zlikovsky, 1998,
https://www.cs.virginia.edu/~robins/papers/The_Moving_Target_Traveling_Salesman_Problem.pdf)

The Agile Manifseto
(Mike Beedle et al, 2001, https://agilemanifesto.org/)

C++
(Bjarne Stroustrup, 1985).

Unreal Engine 4
(Epic Games, Inc., www.unrealengine.com/).

Introduction to Algorithms
(Thomas H. Cormen et al, 1990).

Algorithms
(Robert Sedgewick, 1983),

Further remarks on reducing truncation errors
(William Kahan, 1965)