Albert-Ludwigs-Universität Freiburg
Chair Foundations of Artificial Intelligence
Prof. Dr. Bernhard Nebel

Freiburg, February 1, 2012

Algorithms for the Canadian Traveler's Problem with Remote Sensing

B. Sc. Johannes Aldinger

Abstract

The Canadian Traveler's Problem (CTP) is a stochastic variant of classic graph search. CTP extends the shortest path problem by assigning a blocking probability to each edge. We consider a variant of CTP, CTP with remote sensing, where it is possible to sense the status of edges additionally to moving in the graph. Moving and sensing occur at a cost, and the objective is to reach the goal by keeping the expected total cost as low as possible.

This work investigates state-of-the-art algorithms and presents new approaches to solve CTPs. Especially sampling algorithms like UCT yield good results. Finally, CTP Adapted UCT (CAU) is presented, an algorithm derived from UCT that tackles the problems of remote sensing.

Zusammenfassung

Das Canadian Traveler's Problem (CTP), das Problem des kanadischen Reisenden, ist eine stochastische Variante der Graphensuche. CTP erweitert das Problem der kürzesten Pfade, indem jeder Kante eine Wahrscheinlichkeit zugeordnet wird, mit der die Kante blockiert ist. Wir betrachten eine Variante des CTP, CTP mit remote sensing, bei der es möglich ist, den Zustand entfernter Kanten zu erfassen. Sowohl das Bewegen als auch das Abtasten von entfernten Kanten verursacht Kosten. Die Aufgabe ist es, das Ziel zu erreichen und dabei möglichst geringe erwartete Gesamtkosten zu verursachen.

In dieser Arbeit werden sowohl aktuelle Algorithmen als auch neue Ansätze vorgestellt, um das CTP mit remote sensing zu lösen. Insbesondere Stichproben-basierte Algorithmen wie UCT führen zu guten Ergebnissen. Schlussendlich wird CAU vorgestellt, ein von UCT abgeleiteter Algorithmus der speziell auf die Probleme mit remote sensing eingeht.

Acknowledgements

I give thanks to the members of the Research Group Foundations of Artificial Intelligence in Freiburg. Most notably to Prof. Dr. Bernhard Nebel, who enabled me to write this thesis and to my mentors Patrick Eyerich and Thomas Keller, who took a lot of time to support me in my work. I also want to thank Elisabeth West for checking this thesis for spelling and grammar.

Declaration

I hereby declare that I wrote this thesis on my own, only making use of the sources mentioned.

Freiburg, February 1, 2012 Johannes Aldinger

Contents

1 Introduction

Finding the shortest path in a graph is a common problem with many applications not only restricted to computer science. A famous algorithm to solve the most basic variant of the problem has already been developed as early as 1959 by Edsger Dijkstra [Dij59]. The problem discussed here is a stochastic graph search variant known as the Canadian Traveler's Problem (CTP). It was introduced by Papadimitriou and Yannakakis in 1991 [PY91]. Edges of the graph can be blocked with a given probability which is known by the agent. Once the agent reaches a new state, the conditions of all adjacent edges are revealed.

For larger state-spaces, exhaustive search is not feasible, because the search space is exponential in the number of edges. Recent research has therefore focused on sampling algorithms. Most notably, Upper Confidence bounds applied to Trees (UCT) is a contemporary sampling algorithm developed by Kocsis and Szepesvári [KS06]. UCT yields good solutions on CTP as shown by Eyerich, Keller and Helmert [EKH10].

Finally, in CTP with remote sensing (introduced by Bnaya, Felner and Shimony [BFS09]) it is possible to sense the status of distant edges at a sensing cost. The objective of CTP is to find a path to the goal while minimizing the expected total cost of moving and sensing. CTP can be extended in a way that remote sensing actions can be performed. It is possible to get to know the status of edges not adjacent to the current location at a sensing cost. A promising approach in this context is to estimate an edge's value of information (VoI), a measure for the benefit of knowing its blocking status.

This work investigates common state-of-the-art algorithms for the classic CTP and refines these approaches to solve CTP with remote sensing. The main focus is on CTP Adapted UCT (CAU), an algorithm that extends UCT with several search control techniques. The ideas to enhance UCT are borrowed from various other state-of-the art approaches. CAU uses heuristics and search control techniques originally developed for games like Go, estimates a value of information to guide the search and improves the internal cost estimates with more suitable metrics.

1.1 Applications

The eponymous application of CTP is the problem of a truck driver who wishes to cross the Canadian Rocky Mountains in winter. He has knowledge of the road net, the typical weather circumstances and avalanche probabilities. He wants to reach his target location as fast as possible, but he does not

know which roads are buried by snow. It is debatable if CTP is a good real world model, since its unlikely that the blocking probabilities are independent. Avalanches on one road will likely increase the blocking probabilities of nearby roads, too. Additionally it is not necessarily possible to get to know the blocking status of all roads adjacent to the current village or crossroad. One might need to start moving on a road to finally realize that it is buried.

Like other graph search problems, CTP is more interesting at an abstract level, underlying other tasks that contain subproblems where a shortest path has to be found. This work intends to offer a tool for shortest path problems where non-determinism is involved.

A straightforward approach to tackle CTP is to ignore the blocking probabilities and to assume that all roads are traversable. Such approaches yield reasonable results only if the blocking probabilities are pretty low, which is a decent approach in robot navigation, where obstacles are relatively rare and the detour cost of dodging the obstacles does not exaggeratedly increase the movement cost.

Nevertheless, it is a good idea to put effort into research of problems where the blocking probabilities are *not* negligible. Modern GPS navigation systems could be seen as an instance of CTP with remote sensing, if radio stations charged a fee for revealing the traffic jam information. However, once this information is incorporated into the estimated travel time, the problem becomes deterministic again, which turns GPS navigation somehow into a flawed example.

Another field that could benefit from research in CTP are Internet routing problems. The difference here is that not the edges but the vertices are "blocked" when computers in the network become inoperative. The adaptation is not trivial, but it shows that it is not completely hypothetical to contemplate over non-deterministic shortest path navigation.

Nondeterministic graph search problems that could benefit from CTP or a derived problem can be found in operations research (e.g. a non-deterministic Traveling Salesman Problem) in transportation planning, communication networks, machine learning, and other areas.

1.2 Related Work

The Canadian Traveler's Problem was introduced by Papadimitriou and Yannakakis [PY91]. Exact solutions for special case graphs (directed, acyclic graphs with disjoint paths) have been investigated by Nikolova and Karger [NK08].

Sampling algorithms, in particular UCT [KS06], prove to be a good choice for the CTP. Eyerich, Keller and Helmert successfully applied UCT on the

CTP without remote sensing [EKH10]. In previous work we showed that applying UCT on CTP with remote sensing does not necessarily improve the performance [Ald10] – the larger branching factor is a big problem that leads UCT to degenerate. On rather small graphs the results were still acceptable after a large number of rollouts, but to really improve the UCT baseline more work is necessary. Some of the problems we faced were solved by our CTP Adapted UCT (CAU) algorithm, while other enhancements did not have the impact we hoped they would.

Another approach is the value of information (VoI) estimation of Bnaya, Felner and Shimony [BFS09]. To the best of our knowledge, there is no other work dealing with a CTP variant that allows remote sensing. The movement policy underlying their algorithm is based on Dijkstra's algorithm, which sometimes gives poor estimates. With the help of an estimated VoI this movement decision is improved. Estimating the VoI forms a base for further enhancements and can, for example, also be used as initialization heuristic to estimate the usefulness of sense actions in UCT.

The UCT algorithm has not only been applied to CTP so that search control techniques and ideas from other work on UCT are also useful for our purpose. For example, a former winner of the General Game Playing competition, Cadia Player [FB11], uses MAST and RAVE to enhance its search – two search control techniques that can also be used in our context.

Last but not least an improvement for UCT is model based cost estimate updates and using a different metric to select the children of a search node.

1.3 Focus

This work is centered around sampling algorithms, in particular the UCT algorithm. Exact solutions can only be calculated for small artificial examples by known algorithms. Sampling algorithms approximate uncertainty by averaging over concrete cases. Compared to basic CTP, its remote sensing variant has many more available actions at each decision step. UCT draws its strength from selective sampling to give relevant and promising parts of the search space more confidence. If the search space is too large however, UCT degenerates into vanilla Monte Carlo sampling. Therefore, additional search control techniques have to be developed. Straight-forward UCT implementations do not perform well on CTP with remote sensing and even on CTP without sensing clever initialization techniques are required. Unless the number of samples is huge, it is often best to disallow sensing, since the potential benefit does not outweigh the difficulty to manage the search tree size. In this work we examine ways to improve the performance of UCT. We search for heuristics that restrict the search to more promising regions.

2 Basics

This chapter defines the basics needed to solve the stochastic version of the Canadian Traveler's Problem (CTP). We elaborate on the theoretical background and point out problems that arise when dealing with CTP with remote sensing.

CTP belongs to the class of *deterministic* Partial Observable Markov Decision Processes (POMDP)[1]. The outcome of actions is deterministic as are the observations perceived after each action. The only source of uncertainty comes from the partial observability of the road's blocking status. Like all deterministic POMDPs, the CTP can also be treated as a MDP with an exponentially increased state space, where a state represents the agent's current location as well as the belief of the edges' traversability (unknown, blocked, traversable). In the following section we will formally define the CTP, as well as the basic concepts of the algorithms used in the experimental part of this work.

2.1 CTP

The Canadian Traveler Problem is a graph search problem where edges are traversable with a given probability. The goal is to find a path with *expected minimal cost* from a designated start to a target location.

The blocking probabilities are known by the agent, while the actual status of the roads is only revealed once an adjacent node is visited.

2.1.1 Problem Formalization

Instances of the CTP are 6-tuples $\langle V, E, p, c, v_0, v_\star \rangle$. V denotes the set of *vertices* $\{v_0, \ldots, v_\star\}$ of the graph. The set of *undirected edges* E is a set of two-element subsets of V, connecting two nodes from V. Edges are blocked with a *blocking probability* p. The probability function $p : E \to [0, 1)$ assigns a blocking probability[2] to each edge in E. The travel *cost* $c : E \to \mathbb{N}$ is a function that assigns each edge its respective cost. The vertex $v_0 \in V$ is the *initial node*, and $v_\star \in V$ the *goal*.

The informal representation of the CTP comes from the eponymous scenario of a Canadian truck driver crossing the Rocky Mountains. Therefore, we also use "node" or "location" for the vertices of the graph, while "edges" are also referred to as "roads".

[1] see Lederman Littman [Lit96] for more details on det-POMDPs.

[2] An edge with blocking probability of $p = 1$ does not have to be modeled since it will never be traversable anyways.

2.1.2 CTP with Remote Sensing

The CTP can be extended with remote sensing actions, allowing the agent to get to know the status of edges not adjacent to its current location. The CTP with remote sensing (CTPS) is a 7-Tuple $\langle V, E, p, c, s, v_0, v_\star \rangle$, defined analogously to the CTP without remote sensing. Additionally, a function $s : V \times E \to \mathbb{N}_0$ defines the *cost of sensing* a certain edge $e \in E$ from a given node $v \in V$. In this work we only consider sensing costs $s \in \mathbb{N}_0$ that are fixed for all origin nodes and target edges, so the sensing cost is independent of the agent's current location and independent of the edge to be sensed.

In the examples presented in this work, we annotate edges with $p : c$, the blocking probability followed by the cost. To illustrate many problems we use guaranteed roads with a blocking probability of 0. In this case the blocking probability is omitted. Edges $\{v_i, v_j\}$ are often referred to as edge $v_i - v_j$, which allows an abbreviation when discussing paths, e.g. $v_0 - v_i - v_\star$ instead of $\{v_0, v_i\}\{v_i, v_\star\}$.

2.1.3 Upper Bound

Even though the search space is huge, the number of search nodes visited during a single run is limited. For the CTP without remote sensing, the number of move actions is bounded by n^2 where n is the number of nodes in the graph. It's possible to divide the decision steps of a run into phases. At the end of each phase, the agent moves to a node where new information is revealed. Such a phase is called "macro move". Within that phase, the agent only moves through nodes in a known subgraph. Because Dijkstra's algorithm is optimal, we can request the agent to only move along the Dijkstra path when moving in the known subgraph. In each phase there are at most n nodes that the agent might traverse. The graph has n nodes, so there are at most n such phases, which results in a total number of n^2 visited nodes at most. More precisely, the number of nodes visited is bounded by $\sum_{i=1}^{n} i = \frac{n \cdot (n+1)}{2}$ since in the i-th phase there is a maximum of i nodes in the known subgraph. With remote sensing, the maximum number of possible actions is increased by $|E| - n$. With $|E|$ edges in the graph, there are at most $|E|$ additional actions possible. To retain the worst case of a number of asymptotically quadratic node visits, the movement still has to be done in phases (otherwise, there is only one phase with a total number of $O(n + |E|)$ node visits). Each phase has to end in a node with at least one unvisited edge, so n unvisited edges at the end points of their movement are required.

Consequently, the maximal number of actions in a policy that forbids sensing an edge whose status is already known and that requires to move

along the Dijkstra path when moving within a known subgraph, e.g. the optimal policy, can be upper bounded by $(|E| - n) + \frac{n^2+n}{2}$. The maximal total cost can then be estimated with $c \cdot (|E| - n) + \frac{n^2+n}{2}$ where c is the maximal cost among all edges and sensing costs.

2.2 MDP, Belief State and Policy

In this chapter we introduce Markov Decision Processes (MDP) to be able to define the notion of an action and a policy.

2.2.1 MDP

Markov Decision Processes are a commonly used framework to describe problems where the outcome of actions is stochastic, but the non-determinism only depends on the current state. The outcome of an action is not dependent on decisions made in the past which is also known as the "Markov assumption". Formally, a MDP is a four-tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R} \rangle$ where $\mathcal{S}$ is a set of *belief states*. $\mathcal{A}$ is a set of *actions*, which are mappings $\mathcal{A} : \mathcal{S} \to \mathcal{S}$. Often, $\mathcal{A}$ is also defined as the set of $|\mathcal{S}|$ many sets $\mathcal{A}_s$ which restrict the actions to the actions applicable in the state s. The *transaction probabilities* $\mathcal{P} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0..1]$ are a set of probabilities where $\mathcal{P}(s, a, s')$ is the probability that action a leads from state s to s'. Finally, the set $\mathcal{R} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{Z}$ is the set of *rewards* that occur if action a leads from s to s'.

For the CTP a belief state $s \in \mathcal{S}$ is a combination of the agent's current location v and the belief of the status of each edge in the graph: $s : V \times \{\texttt{blocked}, \texttt{traversable}, \texttt{unknown}\}^{|E|}$. In the following we denote the location v of the agents in state s with $v(s)$. Let also $E_{\texttt{blocked}}(s)$, $E_{\texttt{traversable}}(s)$ and $E_{\texttt{unknown}}(s)$ be the sets of edges containing an edge if and only if that edge is `blocked`, `traversable` or `unknown` in s, respectively.

Actions $\mathcal{A}$ are induced from the CTP definition in the following way: there are two *MOVE* actions for every edge $\{v', v''\} \in E$ of the graph, and one *SENSE* action. However, not all of these actions are applicable in every state. A *MOVE*-action to v'' can only be performed in belief state s, if $v(s) = v'$ and edge $\{v', v''\} \in E_{\texttt{traversable}}(s)$. Analogously, a *SENSE*-actions of edge $e = \{v', v''\}$ can only be applied if the edge in question is `unknown` in the current belief $e \in E_{\texttt{unknown}}(s)$. There are belief states that have no applicable actions at all (when all edge states are known, and all edges adjacent to the current location v' are `blocked`). However, those states will never be reached in an actual run, given good weather.

The transition probabilities $\mathcal{P}(s, a, s')$ depend on the number of outgoing edges of node $v(s')$ and the belief of the status of these edges in state s. The

number of successor nodes is exponential in the number of edges `unknown` in s and adjacent to $v(s')$, since those are the edges that will be revealed once the agent reaches $v(s')$. Every successor s' of a *MOVE*-action in s modifies the agents location to $v(s')$ and the statuses of edges adjacent to $v(s')$ that were `unknown` in s are revealed. The status of all other edges, including edges adjacent to $v(s')$ whose status was already known in s, remains the same in s'. The transition probability $\mathcal{P}(s, a, s')$ is then the product of the blocking probabilities $p(e)$ of all edges that were `unknown` and are `blocked` in s' times the product of $1 - p(e)$ of edges that turned out to be `traversable`. More formally

$$\mathcal{P}(s, a, s') = 1 \cdot \prod_{e \in E_{\mathtt{b}}} p(e) \cdot \prod_{e \in E_{\mathtt{t}}} (1 - p(e))$$

where $E_{\mathtt{b}} = E_{\mathtt{unknown}}(s) \cap E_{\mathtt{blocked}}(s')$ and $E_{\mathtt{t}} = E_{\mathtt{unknown}}(s) \cap E_{\mathtt{traversable}}(s')$[3].

The reward function for the CTP differs slightly from the typical MDP definition. Usually, MDPs are used to maximize a reward, while we use it to minimize the cost of moving and sensing. However, rewards and costs differ only in the algebraic sign. Additionally, the cost of each action is deterministic as the only source of uncertainty comes from the actions applicable in the next state, but the action cost is fixed. Instead of a reward function $\mathcal{R} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{Z}$ we have a cost function $\mathcal{C} : \mathcal{S} \times \mathcal{A} \to \mathbb{N}$. The cost of a move from $v(s)$ to $v(s')$ is the cost $c(\{v(s), v(s')\})$ where $v(s)$ and $v(s')$ are the locations of the states in action a, the vertices of the edge the agent moves along. The objective in the CTP is to minimize the expected cost, which is analog to maximizing the reward.

The initial belief state of the CTP is a state where the agent is at v_0. All edges adjacent to v_0 are either traversable or blocked, while the status of all other edges is unknown.

Goal states are all states s of the search space where the agent's location $v(s)$ is $v_\star$.

2.2.2 States, Actions and Search tree

The task of solving the CTP is to find a way from v_0 to $v_\star$ while keeping the accumulated cost of the applied actions as low as possible. The correct decision depends on the knowledge the agent has of its environment. Moving along a shortest path might be the correct decision most of the time, but once the agent knows that it leads into a blind end, it is advisable to try another way. A state in the search problem CTP is not only the location of the agent, but also its belief of the edge's blocking status.

[3] If both sets $e_{\mathtt{b}}$ and $e_{\mathtt{t}}$ are empty the transition probability is 1.

The actions available at each state are *MOVE*- and *SENSE*-actions. Each *MOVE*-action has a number of possible successors exponential in the number of neighboring edges of the target node, because each edge can turn out to be either blocked or traversable. *SENSE*-actions on the other hand have exactly two possible children, because the edge in question can be either blocked or traversable.

A typical MDP representation are graphs with two different kind of nodes: state nodes and action nodes. A typical search procedure builds a tree, with decision nodes and chance nodes in an alternating manner. Decision nodes refer to the belief states $\mathcal{S}$ of the problem, while chance nodes represent the actions $\mathcal{A}$ chosen. Arcs lead from decision nodes to chance nodes, representing the choice the agent can make. Chance nodes represent the nondeterminism of the system. Its outcome is defined by the transaction probabilities $\mathcal{P}$.

The objective is to find a policy, a mapping of states to actions that minimizes the cost. Since the outcome of actions is nondeterministic it is necessary to give an available action for all states that can be reached in a run under that policy.

2.2.3 Policy

The task of solving the CTP is to find an optimal policy for the problem. A policy $\Pi : \mathcal{S} \to \mathcal{A}_S$ is a mapping from all states in $\mathcal{S}$ to actions applicable in $\mathcal{S}$, so that the expected total cost is minimized. Since the number of states is exponential (there are $|V| \cdot 3^{|E|}$ belief states), it is not necessarily possible to give a compact representation for the policy. Also, there are states in the search space that can never be reached from the initial state and some of them do not even have applicable actions to choose from[4]. Another problem arises from *bad* weather (see Chapter 2.2.4): usually there is a chance that no path from start to goal exists. In those cases all policies are equally good, because the total cost in this run will be infinite no matter what action is chosen. A rational agent might therefore assume that the weather is good. This leads to better results when a path exists in the current weather, while it yields equally bad results if the weather is bad.

2.2.4 Weather

A weather $\mathcal{W}$ is an instance of the CTP which assigns the status `blocked` or `traversable` to each edge $e \in E$. We distinguish between *good* and *bad* weather. A weather is *good* if a path exists from the initial vertex v_0 to the

[4]Because all edges adjacent to the agents location are blocked.

goal $v_\star$ that consists only of traversable edges. If no such path exists, the agent has no chance to reach the goal whatever he does, and the weather is *bad*. Generally, there is a non-zero chance for each graph that the weather is *bad*. The only exception are graphs that contain a path from start to goal where all edges have a blocking probability of 0. This leads to problems when analyzing the expected cost of a policy, because even if the chance that the weather is bad might be small, a run has infinite cost. This in turn leads to an expected cost of infinity of all policies, since the expected cost is the probability-weighted sum of the costs of all possible runs under that policy. Bnaya, Fellner and Shimony [BFS09] solve the problem by introducing a guaranteed high cost edge with blocking probability 0 from start to goal[5]. The solution in this work is to assume that the weather is always good, and to define an optimal policy as a policy with minimal cost, given that the weather is good.

2.3 UCT

Upper Confidence Bound applied to Trees (UCT) is a bandit-based Monte Carlo search algorithm developed by Kocsis and Szepesvári [KS06]. It builds upon UCB1, a bandit algorithm that guarantees logarithmic *regret* (the loss in payout for not always playing the best arm as explained in the upcoming Section 2.3.2) for the multi-armed bandit problem. It became famous due to its application for the game Go [GS07] and has shown to be gainfully applicable in other areas as well.

The motivation for UCT is to focus search on the relevant and "good" actions, while still exploring actions that look suboptimal frequently enough. If it was possible to narrow the relevant actions by $\frac{1}{2}$ at each point in time, the overall work is reduced by a factor of 2^d, where d is the depth of the search tree[6]. If suboptimal parts of the search space can be identified early by UCT because the samples in that area turned out bad, more samples can be used for the relevant parts of the search space.

2.3.1 Monte Carlo Planning

Monte Carlo (MC) methods approximate complex functions by sampling. Given enough samples, the Monte Carlo estimates converge to the actual function.

[5]Which changes the optimal policy as shown in previous work [Ald10].

[6]This motivation for UCT unfortunately also works the other way round: If we only increase the branching factor slightly (e.g. by allowing sense actions), an exponentially larger search space has to be considered.

When MC planning is applied to Markov Decision Processes (MDPs, see Chapter 2.2), a search tree is generated, as proposed by Kearns Mansour and Ng [KMN99]. In an alternating manner decision nodes (state nodes) are followed by chance nodes (state-action nodes).

In decision nodes the agent decides which action to take. The chosen action is a state-action node, also called *chance* node. One of the possible next states is determined depending on the blocking probabilities of the new location's neighboring edges.

A sample in the search tree is called "rollout". During each rollout, a weather is determined but not revealed to the agent. Then, the agent's behavior in this weather is simulated. A big advantage of many sampling algorithms is an "any-time" behavior. An estimate of a state's quality can be given at any time. The more samples are generated, the more precise the result gets.

Unlike classic Monte Carlo sampling, the samples of UCT are not distributed identically among the possible successors, but they are focused on more promising regions of the search space. UCT is based on Upper Confidence Bounds (UCB1), a bandit algorithm by Auer, Cesa-Bianchi and Fischer [ACBF02].

2.3.2 Bandit-Based MC Planning

UCB1 solves the *exploration/exploitation dilemma* for the stationary multi-armed bandit problem. In sequential decision making processes an agent has to find a balance between exploring the search space for potentially better options and exploiting the learned knowledge. If an agent exploits too much he will not find a good solution while an agent exploring too much will find good solutions, but does not use this knowledge to a beneficial effect.

A K-armed bandit is a model for stochastic processes that is inspired by a slot machine with K levers. Formally, we define a set of random variables X_{kn} for each arm $1 \leq k \leq K$ and time step $n \geq 1$. X_{kn} denotes the payoff of playing arm k at time n. For stationary bandit problems all $X_k = \bigcup_{i=1..n} X_{ki}$ are independent and identically distributed with expected values $E(X_k) = \mu_k$. The most beneficial arm is the one with the highest expected value $\mu_\star = \max_{1 \leq k \leq K} \mu_k$. The *regret* is defined as the loss caused by not always playing the best arm $\mu^\star n - \sum_{k=1}^{K} \mu_k n_k$. Here, n_k is the number of plays on machine k up to time n.

For value ranges in $0 \leq X_{kn} \leq 1$ Auer, Cesa-Bianchi and Fischer show that the UCB1 policy yields logarithmic regret. It chooses the arm maximiz-

ing the UCB1 formula:

$$\overline{X_k} + \underbrace{\sqrt{\frac{2 \log n}{n_k}}}_{c(n_k)}$$

where $\overline{X_k}$ is the average reward of playing machine k. The term $c(n_k)$ is a normalization which balances exploration and exploitation. Every time an arm k is pulled, the total number of plays n increases, so the bias term $c(n_k)$ is increased for all but the arm k. For the arm actually pulled, n_k increases, which is in the denominator of the fraction, therefore leading to a decrease of the bias term.

Auer, Cesa-Bianchi and Fischer [ACBF02] prove with the help of Hoeffdings's inequality that the probabilities

$$\mathcal{P}(\overline{X_k} \geq \mu_k + c(n_k)) \leq n^{-4} \tag{1}$$

and

$$\mathcal{P}(\overline{X_k} \leq \mu_k - c(n_k)) \leq n^{-4} \tag{2}$$

are bounded: the achieved average reward $\overline{X_k}$ usually does not deviate from the real expected value μ_k by more than the bias term. Therefore, suboptimal arms are played exponentially less often than the best arm. Auer, Cesa-Bianchi and Fischer also investigate a more sophisticated algorithm UCB2 which lowers the leading constant of the regret near to the theoretical minimum. However, the asymptotic regret of UCB2 remains in $O(\log n)$ and for derived non-stationary problems the simplicity of UCB1 is preferred.

Kocsis and Szepesvári [KS06] transfer the insights of UCB1 to MDPs. Each decision node is treated as a bandit problem. Since the decision of a node deeper in the search tree changes as the agent gathers more information, the bandit problem is not stationary any more. The values of X_{kn} are drifting and are neither identically nor independently distributed. Kocsis and Szepesvári set up drift conditions, under which the regret remains logarithmic, at least for rollouts greater than a lower bound. It can also be shown that the X_{kn} of the MDP converge to the optimal values as the policy converges to the optimal policy.

2.3.3 The UCT Algorithm

Generalizing the UCB1 algorithm on MDPs requires an analysis of non-stationary bandit problems. Obviously, the change in payoff between two time steps must not be arbitrarily large. In UCT, the non-stationary change of the expected outcome of each arm $\overline{X_{kn}}$ converges when the subtree below

the "arm" of the bandit is played "often enough". The logarithmic regret of UCB1 comes from proving the inequalities (see Equation 1 & 2). Kocsis and Szepesváris' main result is that the UCT formula

$$\overline{X_k} + \underbrace{C\sqrt{\frac{\log n}{n_k}}}_{c(n_k)} \tag{3}$$

leads to a non-stationary drifting bandit problem satisfying similar inequalities. The proof is done in an inductive manner over the tree depth, from the leaf nodes to the root. At each step of the induction, a non-stationary bandit problem is faced, similar to that of the UCB1 algorithm. A lower bound N exists on the number of rollouts, so that the expected values $\mu_{kn} = \mathbb{E}(\overline{X_{in}})$ converge sufficiently, meaning that tail inequalities similar to the equations 1 & 2 hold (with the bias parameter $c(n_k)$ of Equation 3).

Leaf nodes do not drift and even have a deterministic outcome, namely the cost from the last edge, and end in the target location. For all nodes higher in the tree a normalization constant C can be found so that for all rollouts after a lower bound N_0 the drift conditions are satisfied. That lower bound becomes greater in each induction step, but remains finite.

The theorems proven by Kocsis and Szepesvári [KS06] also include that suboptimal arms are only played $O(\log(n))$ often. Still, UCT never stops exploring and every action is played infinitely often when the number of rollouts approaches infinity. Eventually, UCT converges to the optimal solution.

These results also hold for discounted infinite horizon problems.

2.3.4 World Model

UCT requires only a *generative* model of its environment. Inner dependencies between random variables need not to be known and one can think of the "sample generator" as a black box. In game playing, samples might be generated by playing against a human opponent, whose behavior does not need to be modeled.

In the CTP, the agent has knowledge of the exact blocking probabilities of each edge. This information is not necessary for UCT, which is because UCT would work as well if the probabilities were not given, but could only be experienced by actually exploring the world. It is bad, because UCT does not take the blocking probabilities directly into account, and considers them only by sampling according to them.

Recall the example of a k-armed bandit: UCT does not need to know the payoff distribution of each arm as it learns from experience and guarantees a

logarithmic regret. However, if the payoff distribution of each arm is known, one can always pull the best one, resulting in a regret of zero. Not using information given by the model results in a suboptimal performance.

2.3.5 Biasing and Initialization

There are different approaches to guide the early rollouts of UCT and similar rollout based algorithms. Some of the more sophisticated ideas are proposed in Chapter 2.5. Since the search space of the CTP is large, a large number of the explored nodes will have very few visits. The initial estimates of a node are therefore important for the algorithms quality.

The UCT formula $\overline{X_k} + C\sqrt{\frac{\log n}{n_k}}$ contains a division by n_k, the number of visits of arm k. In the original bandit problem, an initialization is done by pulling each arm once, thus avoiding a division by zero. In UCT, it is generally not possible to test each path once. If one is not interested in using an initialization, the obvious action selection method would be to choose an unvisited action randomly, if the current node has unvisited children, and if all children have been visited at least once, use the UCT formula. Note that the order of the child visits matters, because visiting a "bad" child with high cost will also give the parent node a worse estimated cost. Therefore, that node will be selected less frequently in future runs. In the limit, the values center around their means, but during actual program execution a bad rollout does have a noticeable effect. In practical applications the performance of the algorithm strongly depends on a clever initialization. The UCT CTP solver of Eyerich, Keller and Helmert [EKH10] achieves good results when initialized with a Dijkstra estimate, but performs very bad when not initialized and randomly choosing children at unvisited nodes. In CTP with sensing a clever initialization is even more important, because the branching factor is much larger. Those initial estimates have to be computable quickly.

Most nodes of a search tree are leaf nodes. In a binary tree half of the nodes are inner nodes, and half of the nodes are leaves, and in a search tree with a higher branching factor like the search tree of the CTP, the percentage of inner nodes is even smaller. This underlines that most of the nodes in the search tree are nodes, that rely on the initialization heavily.

2.3.6 Norm

The value estimated in each UCT node reflects the expected cost of this node. UCT has to aggregate the children costs (including the direct costs to the respective children) to calculate the value of the parent node, and normalize them with the number of total visits. The parent node stores an average

length of the cost, which is composed of the lengths of its children's paths. The natural way to do this is to use the Manhattan norm (1-norm), which sums up all children costs and divides that cost by the number of total child visits. However, it is possible to use other norms to measure that distance.

Basically, all p-norms are reasonable choices. Examples are norms like the Euclidean norm (2-norm) or the maximum norm (∞-norm). Let c_i be the sum of average cost of child i and the direct cost from the current node to this child, and let v_i be the number of visits of child i. The average cost in the p-norm is then given as

$$\sqrt[p]{\frac{\sum_i v_i |c_i|^p}{\sum_i v_i}} = \frac{\sqrt[p]{\sum_i v_i |c_i|^p}}{\sqrt[p]{\sum_i v_i}}$$

A problem with those metrics is that UCT is based on costs rather than rewards. For many purposes this difference does not matter, since costs are just negative rewards, but when dealing with metrics, the triangle inequality demands absolute values. The "best" child is not the node with the highest cost estimate c_i but the one with the lowest. As a simple example assume a parent node with 5 visits, 4 to a node with average cost of 10 and 1 visit to a node with expected cost 20. Applying the Euklidean norm would yield a parent cost of $\sqrt{\frac{4 \cdot 10^2 + 1 \cdot 20^2}{4+1}} = \sqrt{\frac{400+400}{5}} = \sqrt{160} \approx 12.65$ while the cost under the Manhattan norm would be $\frac{40+20}{5} = 12$ which gives more importance to the high cost child, which is the opposite of the desired effect. That problem occurs even more with the maximum norm whose expected cost is $\sqrt[\infty]{\frac{4 \cdot 10^\infty + 20^\infty}{5}} \to 20$.

The reason behind using a different metric is, that the agent can choose its successor in decision nodes and if some exploratory rollout yielded a bad result, that effect should not influence the parent too much. A general solution to this issue is not obvious. In our experiments we recalled the idea and manually implemented a cost estimation similar to the max-norm that estimates the value of a decision node as minimum of its children. With the reasoning that maximizing a reward is the same as minimizing the cost, and the maximum norm does exactly that. However, for other p-norms there is no such conversion. For chance nodes where we cannot influence the outcome, however, and the Manhattan metric might be more appropriate. Another idea is to estimate the values of chance nodes according to the model as described in Chapter 2.3.4.

2.4 Sensing and Value Of Information

Allowing remote sensing increases the practical complexity of solving the Canadian Traveler's Problem. It might seem counterintuitive that solving a problem where the agent has more options, including the options he had before, can be more difficult. The problem is that the agent gets easily overwhelmed by the choice he has, and cannot find the good actions any more, because there are too many options to choose from. Without further effort, allowing *SENSE*-actions deteriorates the algorithm's result, since spread samples equal less samples on the important paths. The advantage of having access to potentially beneficial sense actions is outweighed too often.

2.4.1 Remote Sensing

In Section 2.3.5 we showed that a good initialization is essential for the performance of UCT and similar algorithms. However, it is not obvious how to initialize the sense children. In the next section we will introduce the concept of "value of information" (VoI), which estimates the gain in expected travel cost, once we know the blocking status of an edge. But even given this information, an adequate initialization is not easy to find. Ignoring the benefit of sensing, a first idea is to use the parent's estimate which is the distance from the current location to the goal, and add the sensing cost. However, this approach yields several problems. First, there is no tie-breaker between the many sense actions which can be performed at the current location. This can be avoided when an estimate of the edges VoI is available. A possible initial value would then be the cost from the current location plus the sensing cost minus the value of information.

Low sensing costs lead to a straightforward algorithm like Always Sense (see Chapter 3.1.3) while high sensing costs make it unlikely that sensing is useful at all, resulting in UCT without remote sensing as solid choice. In our experiments we use a sensing cost which is $\frac{1}{5}$ of the average movement cost, which is a value that makes the decision whether sensing is beneficial, hard. Such a sensing cost makes sensing better than accepting detours, while the cost is still high enough to punish careless sensing. Compared to the *MOVE* actions, that sensing cost is rather cheap, and often cheaper than the difference between the estimate of the best and the second best *MOVE* child.

Therefore, UCT often prefers to select sensing a random, useless edge before exploring other *MOVE* actions. The effect is that the algorithm explores a lot of *SENSE* actions but does not find a good path at all. The problem is that *SENSE* children have to have cost estimates comparable to *MOVE*

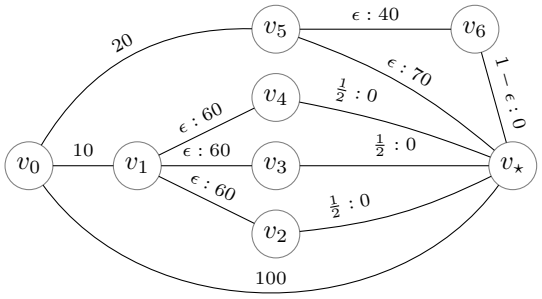

Figure 1: The Graph - a graph to illustrate pitfalls of many approaches

children. In early rollouts, even the true cost of a *SENSE* child is not a good estimate compared to other *MOVE* actions. To illustrate the problem consider the situation depicted in Figure 1: The agent is fooled by the path over edge $v_6 - v_\star$ with a blocking probability of nearly 1, which leads to a promising Dijkstra initialization of 60 in v_0. We refer to this constellation of a cheap but extremely unlikely edge as *Dijkstra trap*. After a few rollouts, the agent realizes that moving from v_5 to v_6 will lead in a dead end, and moving directly to $v_\star$ is the better choice. Assume now, that the sensing cost is 5 and that the above graph is a subgraph of a bigger graph with many `unknown` edges that could be sensed. Sensing one of those edges far away "resets" the initial estimate from the experienced cost of $40 + 40 + 70 = 150$ to $40 + 5 = 45$. In a graph with 200 edges the algorithm would require 400 rollouts to discover the Dijkstra trap, if it would otherwise require only 2. Note that avoiding the problem by initializing the child estimates with the parent's estimate does not necessarily help either – at the time the sense children are created, the parent's estimate was still 40. Also, some sort of pointer to the parent cost comes with problems: if sensing actually *is* useful (which is why we sense in the first place) it is not clear how to decouple the information coming from sensing the edge and the parent cost.

In the end UCT will spend many rollouts on movement decisions, realizing every time that sensing that particular edge is completely useless, but the cost is only increased by a small cost of 5. Instead of selectively sampling relevant parts of the search space, which is the strength of UCT, a lot of samples are wasted.

2.4.2 Value of Information

An agent that has the option to sense an edge is obviously interested if doing so is useful. An approach to measure this usefulness is to calculate the value that the information of the edges status has. The value of information (VoI)

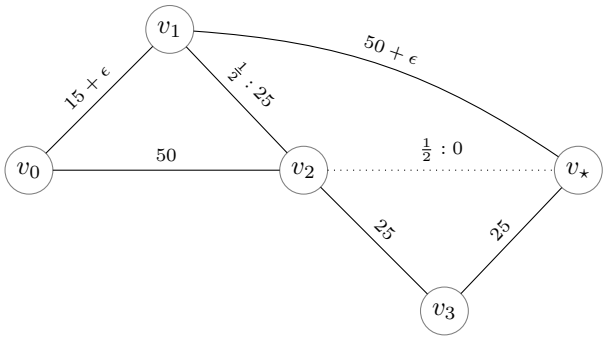

Figure 2: All 4 scenarios NS^+, NS^-, S^+ and S^- are different

of an edge e is defined as difference of expected costs given sensing the edge is possible or not.

$$\mathrm{VoI}(e) = \mathbb{E}(\neg\,\mathrm{sense}(e)) - \mathbb{E}(\mathrm{sense}(e)) \tag{4}$$

The edge in question can be blocked or traversable, which results in four scenarios (cf. [BFS09]):

- $NS^+(e)$: The edge in question was not sensed, and is *traversable*
- $NS^-(e)$: The edge in question was not sensed, and is *blocked*
- $S^+(e)$: The edge in question was sensed, and turns out to be *traversable*
- $S^-(e)$: The edge in question was sensed, and turns out to be *blocked*

Given the expected costs of an optimal policy in these scenarios, the expected value of sensing the edge follow from the edge's blocking probability $p(e)$:

$$\mathbb{E}(\neg\,\mathrm{sense}(e)) = p(e)\mathbb{E}(NS^-(e)) + (1 - p(e))\mathbb{E}(NS^+(e))$$

and

$$\mathbb{E}(\mathrm{sense}(e)) = p(e)\mathbb{E}(S^-(e)) + (1 - p(e))\mathbb{E}(S^+(e))$$

Informally, the VoI consists of two parts: finding shortcuts and avoiding obstacles. Sensing an edge can reveal that S^+ is lower than the cost of the "default" path NS^+. The other benefit is to avoid detours by noticing blocked roads early, in which case S^- is lower than NS^-. A combination of those is also possible, as can be seen in Figure 2. The optimal policy is to sense edge $v_2 - v_\star$ if allowed, and depending on the outcome go to v_1 or to v_2. If the edge may not be sensed, it is best to first move to v_1 and depending on the status of $v_1 - v_2$ go directly to $v_\star$ or dare to move to v_2. The expected

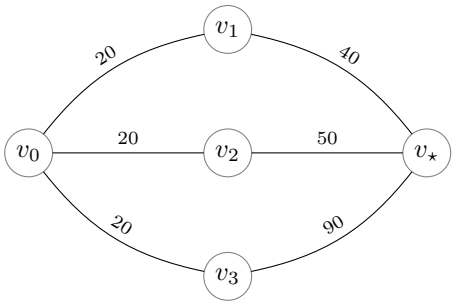

Figure 3: Moving to v_2 is useful, but not optimal

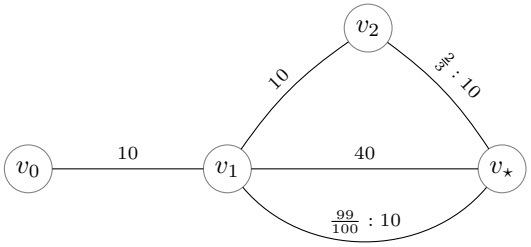

Figure 4: Sensing the edge with the highest VoI is beneficial, but the best action in the initial state is to move to v_1.

values of the dotted edge are $S^+ : 50$, $S^- = 65 + 2\epsilon$, $NS^+ = 52\frac{1}{2} + \frac{3}{2}\epsilon$ and $NS^- = 77\frac{1}{2} + \frac{3}{2}\epsilon$, which results in a VoI of $1\frac{1}{2} + \frac{1}{2}\epsilon$. The detailed calculations can be found in previous work [Ald10].

Once the value of information is known, it is easy to determine whether it is generally advantageous to sense an edge. If the value of information is greater than the sensing cost, $\text{VoI}(e) > c(e)$, then sensing is beneficial. However, not all useful actions are optimal. An evident example consisting only of *MOVE* actions is shown in Figure 3. Moving to v_2 is beneficial, because the expected cost from the current location decreases from 60 to 50. However, there is a better way (namely moving to v_1), which would lead to an optimal path cost. Likewise it might be beneficial to sense a certain edge, but nevertheless not the best thing to do.

Just because an edge's VoI exceeds the sensing cost does not mean it's a good thing to sense an edge immediately. It just increases the likelihood that it is the best action to perform in one of the many belief states. An example where moving is better than sensing, even if the VoI of the edge in question is higher than the sensing cost, can be seen in Figure 4. We presume a sensing cost of 5. It is beneficial to sense edge $v_2 - v_\star$ because its VoI is

$$\mathbb{E}(\neg\,\text{sense}) = \frac{1}{100} \cdot 20 + \frac{99}{100} \cdot 50 = 49\frac{7}{10}$$

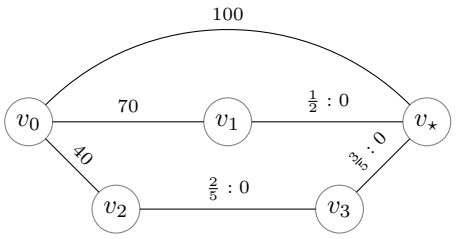

Figure 5: Even though the first action of an optimal policy is to sense an edge, it is not the edge with the highest VoI

$$\mathbb{E}(\text{sense}) = \frac{1}{100} \cdot 20 + \frac{99}{100} \cdot \frac{2}{3} \cdot 50 + \frac{99}{100} \cdot \frac{1}{3} \cdot 30 = 43\frac{1}{10}$$
$$\text{VoI} = 49\frac{7}{10} - 43\frac{1}{10} = 6\frac{3}{5} > 5$$

The VoI of edge $v_1 - v_\star$ is zero. Even though an edge with a VoI higher than the sensing cost exists, the optimal policy would be to first move to v_1 and only in case $v_1 - v_\star$ is blocked, it is optimal to actually sense the edge. Sensing later will reduce the cost by 5 in 1% of the cases for a total expected cost of $\frac{1}{100} \cdot 20 + \frac{99}{100} \cdot \frac{2}{3} \cdot 55 + \frac{99}{100} \cdot \frac{1}{3} \cdot 35 = 48\frac{1}{20}$

A possibly even more surprising fact is that in case the agent knows that sensing an edge will be the best decision, it is not optimal to sense the edge with the highest VoI first. In Figure 5 we can see an example where it is optimal to first schedule sense actions until one path turns out to be traversable and then move along that path. The optimal policy is to sense $v_3 - v_\star$ first. Then, if that edge exists sense $v_2 - v_3$ and move along $v_0 - v_2 - v_3 - v_\star$ if available; otherwise sense $v_1 - v_\star$ and go that way if possible. The expected path cost of sensing in the order $v_3 - v_\star \succ v_2 - v_3 \succ v_1 - v_\star$ yields

$$\tfrac{3}{5} \cdot \tfrac{1}{2} \cdot 110 + \tfrac{3}{5} \cdot \tfrac{1}{2} \cdot 80 + \tfrac{2}{5} \cdot \tfrac{2}{5} \cdot \tfrac{1}{2} \cdot 115 + \tfrac{2}{5} \cdot \tfrac{2}{5} \cdot \tfrac{1}{2} \cdot 85 + \tfrac{2}{5} \cdot \tfrac{3}{5} \cdot 50 = 85$$

while first sensing with priorities $v_1 - v_\star \succ v_3 - v_\star \succ v_2 - v_3$ yields[7]:

$$\tfrac{1}{2} \cdot \tfrac{3}{5} \cdot 110 + \tfrac{1}{2} \cdot \tfrac{2}{5} \cdot \tfrac{2}{5} \cdot 115 + \tfrac{1}{2} \cdot \tfrac{2}{5} \cdot \tfrac{3}{5} \cdot 55 + \tfrac{1}{2} \cdot \tfrac{3}{5} \cdot 80 + \tfrac{1}{2} \cdot \tfrac{2}{5} \cdot \tfrac{2}{5} \cdot 85 + \tfrac{1}{2} \cdot \tfrac{2}{5} \cdot \tfrac{3}{5} \cdot 55 = 86\tfrac{1}{5}$$

The reason is that after sensing $v_1 - v_\star$ it is still beneficial to check whether the cheapest path v_2 might exist.

The VoIs of the three edges of the graph are calculated with the help of the four scenarios NS^+, NS^-, S^+ and S^-. These values are calculated using

[7]For committing policies on the same path it is optimal to first sense the edge with the highest blocking probability. See also [BFS09].

the optimal policy, given the edge in question may not be sensed. However, it is allowed to sense other edges of the graph.

For edge $v_1 - v_\star$ that is:

$$NS^+/NS^-/S^- : \tfrac{3}{5} \cdot 105 + \tfrac{2}{5} \cdot \tfrac{2}{5} \cdot 110 + \tfrac{2}{5} \cdot \tfrac{3}{5} \cdot 50 = 92\tfrac{3}{5}$$

$$S^+ : \tfrac{3}{5} \cdot 75 + \tfrac{2}{5} \cdot \tfrac{2}{5} \cdot 80 + \tfrac{2}{5} \cdot \tfrac{3}{5} \cdot 50 = 69\tfrac{4}{5}$$

$$VoI : \underbrace{\tfrac{1}{2} \cdot 92\tfrac{3}{5} + \tfrac{1}{2} \cdot 92\tfrac{3}{5}}_{\neg\,\text{sense}} - \underbrace{\tfrac{1}{2} \cdot 92\tfrac{3}{5} + \tfrac{1}{2} \cdot 69\tfrac{4}{5}}_{\text{sense}} = 92\tfrac{3}{5} - 81\tfrac{1}{5} = 11\tfrac{2}{5}$$

The value of information for edge $v_3 - v_\star$ is

$$NS^+/NS^-/S^- : \tfrac{1}{2} \cdot 105 + \tfrac{1}{2} \cdot 75 = 90$$

$$S^+ : \tfrac{2}{5} \cdot \tfrac{1}{2} \cdot 110 + \tfrac{2}{5} \cdot \tfrac{1}{2} \cdot 80 + \tfrac{3}{5} \cdot 40 = 62$$

$$VoI : \underbrace{\tfrac{3}{5} \cdot 90 + \tfrac{2}{5} \cdot 90}_{\neg\,\text{sense}} - \underbrace{\tfrac{3}{5} \cdot 90 + \tfrac{2}{5} \cdot 62}_{\text{sense}} = 90 - 78\tfrac{4}{5} = 11\tfrac{1}{5}$$

Finally the VoI of $v_2 - v_3$:

$$NS^+/NS^-/S^- : \tfrac{1}{2} \cdot 105 + \tfrac{1}{2} \cdot 75 = 90$$

$$S^+ : \tfrac{3}{5} \cdot \tfrac{1}{2} \cdot 110 + \tfrac{3}{5} \cdot \tfrac{1}{2} \cdot 80 + \tfrac{2}{5} \cdot 40 = 73$$

$$VoI : \underbrace{\tfrac{2}{5} \cdot 90 + \tfrac{3}{5} \cdot 90}_{\neg\,\text{sense}} - \underbrace{\tfrac{2}{5} \cdot 90 + \tfrac{3}{5} \cdot 73}_{\text{sense}} = 90 - 79\tfrac{4}{5} = 10\tfrac{1}{5}$$

As we can see, even knowledge of the true value of information only helps determining if an edge should be sensed, but by itself is not sufficient to solve the problem and find a good policy. Since exact VoI calculation requires four times the solution of a CTP with only one edge less, it is obvious that it is not practical to try to calculate the VoI exactly.

However, it is a good heuristic estimate and the examples above mentioned show that having access to a VoI helps determining if an edge should be sensed, but because even knowledge of the exact VoI would not solve the problem, it's not crucial that these estimates deviate from the true values from time to time. An algorithm would have to check if it is best to sense the edge in question now or later anyways.

Incorporating VoI estimation as heuristic estimate in another algorithm is therefore advisable, and in this case it is less important that the true VoI can not be calculated and has to be estimated anyways.

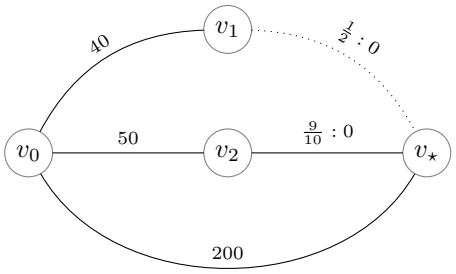

Figure 6: All components of the VOI are different for the edge $v_1 - v_\star$

2.4.3 Disjoint Path CTPs

A subclass of CTPs are disjoint path CTPs. They have the property that each path from start to goal only has two common nodes - start node and terminal node. An interesting observation is that it is possible to construct a disjoint path CTP where all four components to calculate the VOI of an edge are different, so an edge is useful to avoid obstacles and to explore shortcuts at the same time. This is counterintuitive at first, because the edge in question can be either on the intended path – in which case S^+ and NS^+ are equal – or it can be on a path that would only be traversed when we have the certainty that the edge in question exists, but which is not the intended path otherwise. So, one would expect that all NS^+, NS^- and S^- are equal and different from S^+ (sense to find a shortcut) or that S^+ and NS^+ are equal, S^- is an alternative path and NS^- will suffer from a long detour.

However, there is the situation described in Figure 6 where all cases are different. We are interested in the values for edge $v_1 - v_\star$. In this following example we assume a very high sensing cost so that further sense-actions are not part of an optimal policy. If sensing is possible ($S^+ and S^-$) it is optimal to sense edge $v_1 - v_\star$ while otherwise the optimal policy would be to try to move over the short edge. The four scenarios are given as:

- S^+: 40. This is an easy case. We know that the dotted edge exists, so the cheapest path is to move to v_1 and then to $v_\star$
- S^-: 75. The expected cost of moving to v_2 first is $\frac{9}{10} \cdot 50 + \frac{1}{10} \cdot 300 = 66$. In the case that $v_2 - v_\star$ also does not exist, we will directly move from v_0 to $v_\star$ since we already sensed edge $v_1 - v_\star$ and know that it does not exist.
- NS^+: 59. We already know from the last calculation that in case $v_0 - v_1 - v_\star$ does not exist we have an expected path cost of 75. Moving

to v_1 first leads to a cost of $\frac{1}{2} \cdot 40 + \frac{1}{2} \cdot (80 + 75) > 75$, so the best move is to go to v_2 first. Still, moving over $v_0 - v_1$ is better than the direct edge with cost 200, so in case $v_2 - v_\star$ is blocked the correct decision is to try path $v_2 - v_0 - v_1 - v_\star$ (which will then always exist since we are in the NS^+-scenario). The calculation of NS^+ therefore breaks down to $\frac{9}{10} \cdot 50 + \frac{1}{10} \cdot 140 = 59$

- NS^-: 83. In the case $v_2 - v_\star$ is blocked, we try to go over $v_0 - v_1 - v_\star$ again, but this time the path is blocked, so the calculation is: $\frac{9}{10} \cdot 50 + \frac{1}{10} \cdot (100 + 80 + 200) = 45 + 38 = 83$

As we can see, all components have different values even on such a simple disjoint path CTP.

2.5 Search Control Techniques

Compared to other sampling algorithms, UCT draws its advantage from selective sampling. If samples are widespread, the confidence weight on each branch is low, because its estimate is based on few samples only. To achieve good results quickly it is important to preselect promising branches.

UCT is really good at discovering outliers, and will quickly reinforce a node that turned out much better than discovered, as it will also realize fast that a node had an extremely optimistic estimate.

Sensing poses a severe problem to the task, not only because the branching factor is increased generally, but also because sensing an edge seems reasonable, even if it is not beneficial at all. In Section 2.3.5 we reasoned about the problem and in this section we will contemplate sophisticated search control techniques to improve the performance of the algorithm.

Most of the techniques proposed in this section are advanced initialization techniques, but there are other aspects that modify the algorithm otherwise.

2.5.1 SAST

The State Average Sampling Technique (SAST) is a search control mechanism inspired by Cadia Player's [FB11] Move Average Sampling Technique MAST. In game search problems, usually some moves exist that are generally good, no matter what the game state looks like. An example would be placing an "×" in the center cell of Tic Tac Toe. In the CTP there are often the same actions applicable in many states. Especially with macro moves (see Chapter 2.1.3), the benefit of those action greatly varies with the direct cost to move to a certain node. Still, the average distance from that node to the goal is a

good estimate, even though the knowledge of edge states "behind" the road in question do matter.

The adapted idea is to identify nodes in the graph that are favorable independently of the belief state.

The version of UCT of Eyerich, Keller and Helmert uses Dijkstra as initialization method. This comes with the advantage that the value a node is initialized with depends on its belief state. However, it is rather costly to compute because Dijkstra's runtime is in $O(n \log n)$.

SAST is implemented as a table containing two values for each node of the graph: the expected goal distance from this node as well as a number of visits resulting in a measure for the confidence of the expected value. Rollout based methods can easily update this table in the back propagation step.

It might be desirable to secure an independence of the results from the order in which nodes are visited in the search tree. In this case the SAST table should not be updated. Our experiments show that using such a table usually impairs the performance of the algorithm compared to Dijkstra initialization because knowledge about the blocking probabilities of the edges in the current state is abandoned, but this decrease in performance is not significant. On the other hand, gain in runtime is huge. UCT spends around 80% of its time calculating Dijkstra initializations, since the percentage of leaf nodes of a search tree is high. Instead of an $O(n \log n)$ runtime, using a table results in finding initial values in constant time $O(1)$. In the experiments in Chapter 4 we show that the algorithm's performance increases when the same runtime is given, because the slightly worse initialization is more than made up by the constant lookup time.

2.5.2 RAVE

Another initialization technique borrowed from Cadia Player [FB11] is Rapid Action Value Estimation (RAVE). It was originally developed for the game Go. The idea behind RAVE is to assume that a permutation of the available actions may yield similar results, and is therefore a good estimate. A permuted action order is more obvious for *SENSE* actions but when macro moves are used that macro successor has been a valid action at least once in an earlier step. Therefore the "same" *MOVE* action can be applied in from different nodes. The direct costs vary a lot here (for macro moves they are not just the costs of one edge, but the cost of the Dijkstra path from one fringe node to the other), so in order to apply RAVE it is advisable to decompose direct action cost and estimate, and store only the cost achieved in the new location. For *SENSE* actions the benefit of sensing has somehow to be taken into account. If the agent senses an edge while located relatively

near the goal, the estimated benefit is not the cost from the location to the goal, but rather some estimate on the VoI (see Section 2.4.2) of the sensed edge. An easily accessible VoI estimate is the difference of the estimate of the best move child and the estimated cost after sensing the edge in question.

A rollout in the algorithm contains two phases. In the first phase the next action is chosen – usually according to the UCT formula. Then, the chosen action is performed, and recursively a rollout of the next node is made. When this rollout has finished, the cost from the current node to the goal is returned. With RAVE, all actions made on the way to the goal are also returned. This can be implemented with the help of a RAVE-set that is forwarded to the children, which append the current action to the set. During the back propagation step it is checked whether one of those action would have been applicable in the current state. If so, the cost from that node can now be taken as estimate for the children that were not visited in the current run.

However, there are examples where the order of actions *does* matter. Therefore, the RAVE estimates should only be considered as long as the number of rollouts is low. To achieve this there is a way to modify the estimated costs within the UCT formula. An equivalence parameter k specifies the number at which the RAVE estimate has the same influence in the UCT formula as the experienced cost.

Each search node stores two $\mathcal{Q}$ values: the normal $\mathcal{Q}(s,a)$ estimates and with RAVE additionally $\mathcal{Q}_{RAVE}(s,a)$ the values coming from the back propagation step in the rollouts.

Instead of the old UCT estimate the RAVE estimates are computed as

$$\beta(s) \cdot \mathcal{Q}_{RAVE}(s,a) + (1 - \beta(s)) \cdot \mathcal{Q}(s,a)$$

where

$$\beta(s) = \sqrt{\frac{k}{3 \cdot N(s) + k}}$$

It can be seen that the relevance of the RAVE value is initially high, with k rollouts identical and then diminishes quadratically.

The RAVE estimation can also be used to initialize the algorithm, without influencing actually experienced costs. Instead of adding virtual visits to the search nodes themselves, those initial values can be stored in the RAVE estimation which automatically cares for a diminishing influence of these artificial values.

2.5.3 Sense Fees

Another problem is the initialization of *SENSE* actions. Once the algorithm has converged and each node contains its true cost, completely useless sense actions will have an estimated cost, which is the cost of the best actions the agent can take plus the cost of sensing the edge. In our experiments we used a sensing cost of 5 which is high enough to punish exaggerated sensing and yet low enough that chances are high that there is one or another edge in the graph that is actually worth sensing (see also Section 2.3.5).

To avoid that sense children get preferred over move children in general, we propose two ideas that both artificially increase the estimate for sense successors.

Sense Fees increase the initial estimate of sense children with an exaggeratedly high cost. In the first rollouts sensing seems therefore so bad that early rollouts will consider *MOVE* actions only. However, the UCT formula is designed in such a way that rollouts that turn out to be much better than their original estimate will be tried more often because they still have a high UCT bonus. This results in a behavior where the algorithm moves until at one point it tries its first *SENSE* action. At this point the algorithm will basically continue sensing, until a situation similar to the algorithm without sense fees is reached. It is hard to find a sense fee where the algorithm's behavior does not correspond to the algorithm without sensing (sense fee too high) or the exaggeratedly sensing UCT with Remote Sensing (UCTRS) (sense fee too low) which is a straightforward extension of UCT implemented in former work [Ald10].

Another approach is to permanently punish bad sense children. The problem of sensing is that in the early rollouts, sense successors are preferred compared to move successors. Sensing an edge that does not improve the expected cost of the node is less beneficial for exploring the search tree than investigating the worst available move. However, the estimate of the useless sense child is usually even better than the estimate of the second best move. To adjust this, one can simply compare the estimate of the bad sense child with the worst move instead of the best move.

In our experiments we estimate the cost of sensing as direct sensing cost plus cost of the best move minus estimated VoI of sensing that edge, if the VoI is larger than the sensing cost. If the VoI is smaller, then the cost is estimated as direct sensing cost plus estimate of worst move minus value of information. In other words, if the estimated VoI of an edge is smaller than the sensing cost, the cost difference between worst move estimate and best move estimate is added to the estimated cost of the sense action.

This is a severe modification to the UCT formula, since many sense ac-

tions will obtain much less rollouts than they otherwise would. If the values of UCT converged to the true estimates it *is* better to sense a random edge, than to move to a location further away from the goal. If sensing an edge really is beneficial however, its cost will be compared with the best move child again. We assume that the values still converge to the optimal policy, since also suboptimal children are visited infinitely often. However, proving it is far from trivial. In the experiment 5 in Chapter 4 we can see that this approach is good, despite its theoretical weakness.

3 Algorithms for the CTP

In this chapter we propose different approaches to solve the Canadian Traveler's Problem. Some of the algorithms presented here solve only the CTP without remote sensing, but for most of them we present an extension. At the end we present the CAU algorithm, which combines the benefits of the approaches proposed before.

3.1 Dijkstra variants

The most straightforward solution for the CTP is to use an adaptation of Dijkstra's Algorithm [Dij59]. Its runtime is in $O(e \cdot n \log n)$ where e is the number of edges of the graph and n is the number of nodes. It can find the shortest distance from all nodes in the graph to the goal.

Dijkstra manages a list of open nodes in a priority queue (open list) ordered by distance to the target location. A closed list as list of the nodes already visited stores the nodes whose shortest distance is already known. Initially, the closed list is empty, and the target node is put into the open list (with a cost of 0).

During each step of the algorithm, the node with the lowest cost is popped from the open list and added to the close list, if no cheaper entry has been stored there already. All of this node's neighbors are then added to the priority queue with the direct cost added to the current cost.

3.1.1 Optimistic Dijkstra

To apply Dijkstra's Algorithm to the nondeterministic graph search problem CTP, dealing with blocked roads is required. The most straightforward approach is to ignore the blocking probabilities completely, and assume that all edges are traversable. This is also known as the *free space assumption.*

Acting under the free space assumption is feasible if the blocking probabilities are low. However, if the traversability of the shortest path has a low probability, the performance of Dijkstra can become extremely bad. The example (taken from [EKH10]) from Chapter 2.4.1 that illustrates several weaknesses of various algorithms is shown once more in Figure 7.

In Figure 7 there are some roads with a small blocking probability of $\epsilon \gtrapprox 0$, and one edge $v_6 - v_\star$ which is almost always blocked.

Dijkstra's Algorithm will move along the path $v_0 - v_5 - v_6$ because it is mislead by the low cost of the edge $v_6 - v_\star$ and estimates the movement cost of the initial state with a total cost of $20 + 40 = 60$. However, at v_6, the edge

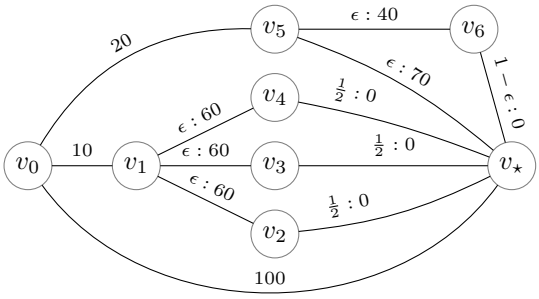

Figure 7: The graph

turns out to be blocked, and so from v_6 the agent will continue $v_6 - v_5 - v_\star$ for a total cost of 170.

Nevertheless, Dijkstra offers a great baseline, and when the blocking probabilities are low the algorithm performs relatively well. Because of its relatively fast runtime, Dijkstra can also be used in many other algorithms as a reasonable heuristic estimate.

3.1.2 Normalized Dijkstra

It is possible to consider the blocking probabilities while not losing the benefit of a fast estimation. The trick is to normalize the edge costs with the inverse probability that the edge is traversable. The normalized cost c_n is given as $c_n = c \cdot \frac{1}{1-p}$ where p is the blocking probability.

Given these normalized costs there are two possible ways to use them. The first possibility is to replace the normalized costs c_n with the actual costs. The algorithm does not change at all and is just run with modified edge costs. The second possibility is to store two estimates for each node in the open list. The priority is based on the normalized cost, while the cost written back to the closed list remains the original path cost. The advantage of this method is that the estimated cost comes from an actually possible run through the graph. So the normalization modifies the selected path and not the cost of that path as well.

Those normalized Dijkstra estimates obviously do not reflect the true expected cost. The actual cost of a detour might be negligible if there are parallel edges with similar costs. And the detour cost can be exhaustively large if no other way exists, in which case also normalized Dijkstra is too optimistic (see also Chapter 2.3.5, optimistic walk-around).

In the Graph in Figure 7 both normalized variants would suggest one of the paths $v_0 - v_1 - v_2 - v_\star$, $v_0 - v_1 - v_3 - v_\star$ or $v_0 - v_1 - v_4 - v_\star$. Since the real detour cost of the last edge is not considered the algorithm is mislead

and will move over one of those paths for an expected cost of

$$
\begin{aligned}
& \tfrac{1}{2} \cdot 70 && \text{(if } v_2 - v_\star \text{ is traversable)} \\
+ \quad & \tfrac{1}{4} \cdot 190 && \text{(if } v_3 - v_\star \text{ is traversable and } v_2 - v_\star \text{ is blocked)} \\
+ \quad & \tfrac{1}{8} \cdot 310 && \text{(if } v_4 - v_\star \text{ is traversable, } v_2 - v_\star \text{ and } v_3 - v_\star \text{ are blocked)} \\
+ \quad & \tfrac{1}{8} \cdot 530 && \text{(if } v_2 - v_\star, v_3 - v_\star \text{ and } v_4 - v_\star \text{ are blocked)} \\
= \quad & 35 + 47\tfrac{1}{4} + 31\tfrac{1}{4} + 66\tfrac{1}{4} && = 179\tfrac{3}{4}
\end{aligned}
$$

Which is even worse than the regular Dijkstra estimate in this example. In our experiments both variants of normalized Dijkstra turned out to be better initializations than regular Dijkstra, but the difference is small.

3.1.3 Always Sense

A straightforward approach to extend the aforementioned Dijkstra algorithm with the ability to remotely sense some edges, is Always Sense (AS). AS senses all edges along the Dijkstra path until that path is either safely traversable, or an edge is blocked, in which case the optimal Dijkstra path is calculated with the updated weather and the whole procedure restarts. This way the movement cost is reduced to the minimal cost possible under the current weather.

The order in which the *SENSE* actions are performed matters. Bnaya et al. [BFS09] show that it is best to first sense the edge with the highest blocking probability. This is also intuitively reasonable since the agent then detects unlikely paths more quickly and will not spend actions to sense on that path only to discard the whole path later.

Experimental results show that the algorithm usually performs worse than regular Dijkstra which is no surprise. (see Experiment 1, Chapter 4) The information gain of edges with very low or very high blocking probabilities are rarely worth the sensing cost. Nevertheless AS becomes optimal if the sensing costs converge to zero.

3.2 Policy Iteration

The normal CTP without remote sensing can be solved with an adapted Policy Iteration (PI) algorithm known from reinforcement learning that can be applied to solve MDPs. Since the belief space is exponential in the graph size and PI updates all nodes of the search space at every step, the normal Policy Iteration algorithm is no feasible solution for CTP. We presented here a modification on a reduced search space, where search nodes are the direct

nodes of the graph. After one step is performed, and new information of adjacent edges is revealed, a new problem is faced. Still, the estimates of the last run can be used as feasible initial values in the next run.

Policy Iteration is an algorithm consisting out of two phases that decouple the optimality assignment of the Bellman Equation

$$V_{t+1}(s) = \underset{a}{\operatorname{argmax}} \sum_{s'} T(s, a, s') \cdot (c(s, s') + V_t(s'))$$

where $V_t(s)$ is a *value function* dependent on time step t that estimates the utility of being in state s, $T(s, a, s')$ is the *transaction probability* that action a will lead from s to s' and $c(s, s')$ is the cost of edge $\{s, s'\}$. In the first phase, the values are estimated and in the second phase the policy is updated.

$$V_{t+1}(s) \leftarrow \sum_{s'} T(s, a_p, s') \cdot (c(s, s') + V_t(s'))$$

In the first phase the policy is fixed and the best action is already determined for each state. The values of all nodes in the search graph are iteratively updated, until they converge.

$$a_p \leftarrow \operatorname{argmax} \sum_{s'} T(s, a, s') \cdot (c(s, s') + V_t(s'))$$

In the second phase the policy is updated to the best[8] policy given the values calculated in the first phase.

The original Policy Iteration algorithm performs the first phase until the values have converged; however, it is possible to only iterate the values for some steps. In our experiments we used an approach where only one step of the value iteration phase is done, followed by a policy update step.

At each step within each of the two phases, *all* nodes of the search space are updated once (in the first phase the value is updated, in the second the policy). Unfortunately, the search space of the CTP is exponential (with a base of 3) in the number of edges, which are themselves up to quadratic in the number of nodes. It is easy to understand that it's not feasible to update such a big search space even only once, much less to iterate values until convergence.

The trick to apply policy iteration is to modify the notion of a "state" and an "action". Instead of a belief we base Policy Iteration on the nodes of the graph. But since the applicability of a move depends on the edge's

[8]The "best" action in this context is the action with the lowest cost. argmax is therefore a bit misleading since it specifies the action with minimum cost.

blocking status, the notion of an "action" has to be redefined as well. An action $a(s)$ in the Policy Iteration algorithm consists of a permutation of the move actions applicable at the state s, and is a preference ordering, which move the agent would choose, if the corresponding edge exists.

$$a(s) = a_1(s) \succ a_1(s) \succ \cdots \succ a_n(s)$$

The number of those preference-actions is still exponential to the number N of outgoing edges of the corresponding node, but in the value iteration phase the policy is fixed to one of those actions. In the policy update phase the optimal action has to be selected among exponentially many actions, but the best one can easily be identified by looking at the cost estimates of neighboring states. The best policy is the one that moves to the cheapest available neighbor, so it is just an ascending ordering of movement cost plus neighbor's estimate.

For the algorithm we assume that all nodes are reachable by at least one of their neighbors. The reasoning is that a node that will be visited during an actual run can at least go back to the location where it just came from. The *MOVE*-actions a_i have an associated probability p_i which is the probability that the edge to move from s to s_i exists. All inner actions applicable in a state will result in different successor states and each of them can only be reached by exactly one action.

The N actions a_i available at a certain node are ordered by the preference relation so that the index $i \in [1 \dots N]$.

This leads to the following transition probabilities:

$$T(s,a,s') = \begin{cases} \underbrace{\prod_{i}^{i<j} (1-p_i)}_{\text{probability that another edge was chosen first}} \cdot \underbrace{p_j}_{\text{edge chosen}}, & \text{if } j < N \\ \underbrace{\prod_{i}^{i<j} (1-p_i)}_{\text{last edge always exists}}, & \text{if } j = N \end{cases}$$

Let the sum $E_i = c_i + V(s_i)$ be the total estimated cost (including direct cost) of moving to s_i. Then the new estimate is given as

$$p_0 \cdot E_0 + (1-p_0)p_1 \cdot E_1 + \cdots + (1-p_0)(1-p_1)\dots(1-p_{N-1}) \cdot E_N$$

After each actual move, new knowledge is incorporated into the state, because edges adjacent to the new location turn out to be blocked or traversable.

At this point in time, the algorithm has to be restarted with adjusted transaction probabilities. Remote Sensing CTP is not feasible for the Policy Iteration algorithm. If a certain edge should be tested, the algorithm would have to be run three times: once with that edge blocked, once with that edge traversable and once with that edge left unknown. After all three scenarios have converged, the estimates of the initial state show if sensing the edge would be beneficial or not. However, Policy Iteration is rather an alternative for regular CTP.

3.3 VOI and EXP

Bnaya, Felner and Shimony were the first to consider the CTP with remote sensing [BFS09]. They introduce two algorithms, EXP and VOI[9], which both rely heavily on the Dijkstra Algorithm (see Chapter 3.1) and use remote sensing to alleviate the traps that Dijkstra falls into.

Edges are sensed if their value of information (VoI, see Chapter 2.4.2) exceeds the sensing cost. The correct estimation of the value of information is infeasible and has to be approximated. Bnaya, Felner and Shimony also restrict the *MOVE*-actions to committing policies that may not depart from an intended path unless an edge on that path turns out to be blocked[10].

Initially, the intended path is set as the Dijkstra path from start to goal. VoI is only checked for edges on this path. Since the agent will not change its behavior if the edge is traversable, the S^+ and NS^+ cases fall together, which simplifies VoI calculation. Therefore, only the "sensing to avoid obstacles"-part of VoI is considered.

To estimate the VoI, the assumption that all edges are traversable except the edge in question is made. In the EXP algorithm, S^- is calculated as Dijkstra cost from start to goal in an optimistic graph where all but the edge in question are traversable. NS^- is the cost to first move along the intended path to the start of that edge plus the optimistic Dijkstra cost from there to the goal. The action with the highest VoI determined this way is then sensed, if its VoI is higher than the sensing cost. Otherwise, the agent starts moving.

The VOI algorithm refines the VoI calculation of EXP. It is no longer assumed that all edges are traversable. Instead sampling is performed to estimate the value of information. The way expected movement costs are

[9]We use VoI as abbreviation for "value of information" and VOI (all capital letters) for the algorithm by Bnaya et al.

[10]Committing policies are just a partial action assignment to all states in which the current node is on an intended path, and the action leading to the next node on that path is applicable.

estimated is similar to that of the ORO algorithm which will be presented in Chapter 3.4. Instead of assuming that all edges are traversable, a weather (see Chapter 2.2.4) is generated. An agent using the optimistic Dijkstra policy traversing that weather is simulated. This process is repeated several times, and the average path cost is taken to estimate S^- and NS^- for the VoI estimation step. However, the "intended path" remains the optimistic Dijkstra path, and sampling is not used to improve the movement.

3.4 HOP and ORO

Eyerich, Keller and Helmert introduce two rollout-based algorithms for the CTP without remote sensing to improve the movement decision [EKH10].

Hindsight Optimization (HOP) is an approach that is also known as "averaging over clairvoyance". During each rollout, the agent simulates a weather and calculates the Dijkstra distance from the current node to the goal with full information of that weather. The average cost of those rollouts is used to determine the estimated movement cost.

In the graph depicted in Figure 7 the agent would first move to v_1 because chances are high that one of the edges $v_2 - v_\star$, $v_3 - v_\star$ or $v_4 - v_\star$ will exist, and because of the clairvoyance assumption the agent knows the correct successor. However, once v_1 is reached, HOP realizes that the information which one(s) of those three edges actually is traversable is missing, and would proceed to the goal over $v_1 - v_0 - v_5 - v_\star$ for a total cost of 110.

Optimistic Rollout (ORO) is a representative of a family of rollout-based algorithms. During each rollout the algorithm does not assume clairvoyance, but instead simulates an agent acting under a certain policy. That policy has to be quickly calculable, since it is used at each visited node once in each rollout. The simulated agent's behavior in ORO is the Optimistic Dijkstra algorithm.

In the example of Figure 7 ORO would be fooled by the exaggerated cost the simulated agent suffers from the "Dijkstra trap" $v_6 - v_\star$ and would therefore move directly to $v_\star$ for a cost of 100. ORO can also be seen as a Dijkstra-initialized UCT algorithm that never explores, but only exploits during each rollout.

Extending these algorithms with remote sensing actions is awkward for HOP, since the clairvoyance assumption makes it hard to incorporate the benefit that sensing could have. ORO is much more suitable to be extended with remote sensing actions. A combination of Bnaya, Felner and Shimony's VOI algorithm and ORO is conceivable, even though we did not try to implement it, as UCT is a more sophisticated sampling algorithm.

3.5 Table Based CTP Solver

A problem of Bnaya, Fellner and Shimonys VOI algorithm is that the movement decision is based on the Dijkstra path even if that path is unlikely. Additionally, the value of information an edge can have consists of two parts: avoiding obstacles and detecting shortcuts. The motivation of Table Based CTP Solver (TBC) is to find an algorithm that uses both parts of the VoI. Like ORO, the TBC algorithm averages over the cost of a simulated agent. TBC uses tables to store the average path costs from any point in the graph to the goal. These are used to guide the simulated agent as well es to calculate the VoI. In the end, due to some theoretical weaknesses the algorithm does not perform good in practice, and tends to sense a lot of unnecessary edges, which leads to the worst performance of all considered algorithms. However, the reason behind this bad behavior gives interesting insights into the CTP with remote sensing.

TBC essentially consists of four tables that store different cost averages: StartPresent, StartMissing, GoalPresent and GoalMissing of size $|N| \cdot |E|$ where $|N|$ is the number of nodes in the graph and $|E|$ is the number of edges. For each edge e, the tables store the average path cost of a node to the start or goal, given e is traversable or blocked, while all others are sampled according to their blocking probability.

The TBC algorithm determines its sensing decision under the assumption that it is possible to sense exactly once, and then start moving. However, after submitting an action to the server, the algorithm is restarted with the updated belief. Therefore, it is possible that the final agent's behavior alternates between sense and move actions, even if the internal estimates do not.

The algorithm performs a number of rollouts to estimate the expected distance from each node in the graph to the goal. The estimated cost of a *MOVE* neighbor could be calculated from the GoalPresent and GoalMissing tables, e.g. by taking an edge known to be present and lookup the distance from the neighbor to the goal, given that edge is present. However, the confidence into a single estimate is not extremely high. Therefore, the TBC algorithm additionally stores a table AccumulatedGoalTable that contains the average cost from each node in the graph to the goal, averaged over all weathers encountered so far. The estimated cost of a node's neighbor is then the direct cost plus that neighbor's estimate in the accumulated goal table. The estimate of that table can also be calculated out of the two goal distance tables as the average of average path costs. For each edge the estimated distance is calculated as the probability that the edge is missing times the value in the GoalMissing table plus the probability that the edge is

present times the value in the GoalPresent table. If the table entries would be consistent these values should be equal for all edges. But the single table entries obviously represent only estimates and not true costs. To achieve the estimate with the highest confidence, the accumulated goal distance is then the average of these estimates.

In Chapter 2.4.2 we introduced four components to calculate the value of information of an edge: NS^+, NS^-, S^+ and S^-. These can be calculated from the four tables with two restrictions. Optimally, the scenarios refer to the expected costs of an agent acting under the optimal policy, where sensing all but the edge in question is possible. The values in the tables try to approximate the expected distance of an agent that will only move, so further sense actions are not part of that policy. The second restriction is that the table entries of TBC do not converge to the true expected costs because all rollouts are goal-oriented and causes problems that make the estimates of the StartPresent and the StartMissing table be far off their true values, as we will investigate later. Even if the table entries would correspond to the true costs, the resulting policy would not have to be optimal. In the example in Figure 5 we showed that it does not have to be optimal to sense the edge with the highest VoI first, even if sensing an edge is correct. Even more, the assumption of TBC that sensing can only be performed once before moving, makes finding the correct decision even harder. Still, the emerging policy should be better (or equal) than the optimal policy for the CTP without remote sensing. The table entries approximate solutions for special cases of the CTP without remote sensing, and this policy could only be improved by sense actions where the value of information is higher than the sensing cost.

Still, given the tables would contain the actual expected distance between a node in the graph and the start or goal, it is possible to calculate the VoI of an edge e under a subsequent move-only policy. The value $NS^+(e)$ is directly available twice in the tables: in the GoalPresent table it's the entry for v_0, e where v_0 is the current position, and in the StartPresent table its $v_\star, e$, which both contain the expected path length from start to goal, given edge e is present. In the algorithem the entry from the GoalPresent table has a higher confidence and corresponds more to the true value, because more information from previous steps can be reused.

For the NS^- case the same observations holds, and the entry is stored both in the StartMissing and in the GoalMissing table.

To get an estimate for S^+ one may assume that the edge in question does exist. Let the edge in question be $e = v_i - v_j$. Since it is not clear in which direction to traverse the edge, S^+ is the minimum of $\text{StartPresent}(v_i, e) + c(e) + \text{GoalPresent}(v_j, e)$, $\text{StartPresent}(v_j, e) + c(e) + \text{GoalPresent}(v_i, e)$ and NS^+. Just because the edge is present, the agent does not have to move over

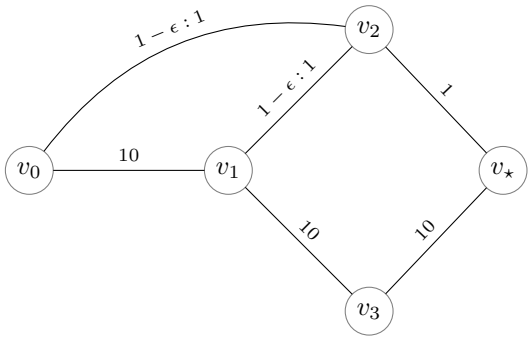

Figure 8: The best path is so good, because it only considers short, but unlikely paths.

the edge, but if it is present he may do so.

The most difficult case is S^- since it is not obvious how to avoid a potential dead end road if the agent knows that the edge is blocked. The idea is that there exists a node in the graph which leads the agent around the obstacle. This waypoint can be any node in the graph. To find the best one, for all nodes v_w in the graph the path cost $\text{StartMissing}(v_w, e) + \text{GoalMissing}(v_w, e)$ is calculated, and $S^-(e)$ is set as the minimum of those values.

There are different ways to initialize the tables, which usually go back to Dijkstra estimates in some form. Either the shortest distance in a possibly normalized (see Chapter 3.1.2) Dijkstra can be used directly, or, more like ORO, an agent acting under a Dijkstra policy can be simulated. The estimation of the actual values is then done in a number of rollouts. At the beginning of the rollout a weather is simulated that assigns a blocking status to each edge whose status is unknown in the current state. An agent is simulated, using the AccumulatedGoalTable as heuristic. In our implementation we always chose the best action and never explore suboptimal branches, baring in mind that some sort of exploration occurs due to non-existing edges. To avoid loops and to ensure termination, macro moves as described in Chapter 2.1.3 are used. Each edge that was visited in the current run from start to goal is memorized, along with the cost from each node. At the end of the run all tables are updated, for each node on the path, and each edge memorized.

However, a phenomenon occurrs that leads to strange estimates especially of the S^- table. In Figure 8 we see the reason why the estimates in the table do not correspond at all to the actual cost. The initial estimate of v_2 is pretty good, since its goal distance is only 1. Let $\epsilon \gtrapprox 0$, so that the edges $v_0 - v_2$ and $v_1 - v_2$ have a high blocking probability. In every single rollout in which one of those edges is traversable, the agent will therefore go over v_2, because it *is* the best choice. However, if those edges do not exist, the agent

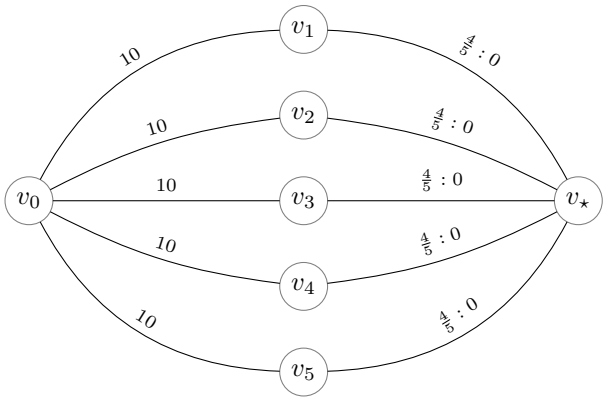

Figure 9: The forward and backwards expected distance of to nodes may differ arbitrarily.

will never consider to move over v_2. In the estimation of the S^- table this leads to a large estimate for the waypoint v_2. However, the StartPresent and StartMissing table entries for v_2 do not correspond to the expected cost to move from v_0 to v_2 as they should, because all rollouts are "goal-oriented", meaning that the agent does not try to move to v_2 but is only intended in moving to $v_\star$.

Improving StartPresent and StartMissing by really trying to move to v_2 is not feasible either, because in the end this would require an own table for each node in the graph, and solving a new CTP for each of those nodes.

A feigned solution to the problem is to introduce backwards-rollouts were Start and Goal tables are exchanged. If v_2 has such a good estimate, the agent will try to reach this node in the backwards rollout, will realize that the edge $v_0 - v_1$ is blocked and have to go back over $v_2 - v_\star - v_3 - v_1 - v_0$ which will increase the cost estimate of going from v_0 to v_2. When rollouts are made forwards and backwards in an alternating manner, the cost estimates would converge to much more realistic values, if the expected shortest distance from v_i to v_j would be symmetric. Unfortunately, this is not the case, as illustrated in Figure 9. Given the weather is good, the expected shortest path from $v_\star$ to v_0 is 10. The agent can choose one of the (few) edges with high blocking probability, and move along that edge. The opposite direction is much more problematic. The most likely case is that exactly one of the 5 edges over v_1, v_2, v_3, v_4 or v_5 is traversable. An agent trying to move to $v_\star$ will find a traversable edge after expectedly half of its tries. The expected travel cost is $2 \cdot 10 = 20$ for all edges that led into a dead end up to the successful try with a cost of 10. So the expected average cost for moving from v_0 to $v_\star$ given exactly one edge is traversable is 50. With the good weather assumption at

hand, the probability that all edges are blocked is zero, and the probability that two or more edges exist is greater 0, so the first traversable edge will be found slightly before on average. This reduces the cost a bit, but does not alter the fact that forward and backward distance vary a lot. With n parallel edges that all lead to a dead end road with a blocking probability of $\frac{n-1}{n}$ the movement cost is n times higher in one direction than in the other. And even if this difference does not scale linearly with the good weather assumption, it still differs arbitrarily.

This in turn makes backwards rollouts not converge to the true cost. We did not find a way to find realistic estimates for the table entries. A problem with this is that once such a inconsistency exists, the sensing decision is based on it and will estimate artificial values of information that do not correspond to their true value. In the end the algorithm starts sensing a lot, and performs even worse than the "Always Sense" algorithm that only senses along the Dijkstra path.

3.6 UCT

Upper Confidence Bound applied to Trees (UCT) has recently become the most famous sampling algorithm due to its success in game playing and many other areas. The fundamentals of the algorithm are described in Chapter 2.3. In this chapter, we elaborate on the implementation details.

In game problems, the search procedure for UCT is often classified into four phases: selection, expansion, simulation and back propagation. During the selection phase the known search tree is traversed. Here the UCT formula is applied to select the next successor. Once a previously unvisited search node is reached, the search tree has to be expanded and a new search node has to be created. The simulation phase will now estimate a reward for the newly created search node. Finally, the achieved result is propagated back to the root and all nodes on that way are updated with the new information.

For the CTP we do not separate the expansion and simulation phase, but instead create search nodes until a goal state is reached. To guide the creation of new search nodes, some sort of initialization is required. The approach of Eyerich, Keller and Helmert is to include a number of virtual visits to each node, that are calculated by estimating the optimistic Dijkstra distance from that node to the goal [EKH10].

In Chapter 2.5.2 we also introduce the possibility to store initial estimates in an extra field of the search node, to better decouple the initial value from the costs experienced. The initial values can be estimated with different heuristics, like optimistic Dijkstra, normalized Dijkstra, and variants like optimistic walk-around that are presented in Chapter 2.3.5.

The exact structure of the search tree can also be implemented in different ways. The basic structure is a compound of decision nodes and chance nodes in an alternating manner. Decision nodes represent the available actions of the agent, while chance nodes contain the effects of those actions. Decision nodes can be modified in that not the direct *MOVE* neighbors are available, but all nodes at the fringe of the known part of the graph. The use of such macro moves (see Chapter 2.1.3) is motivated by the maximum number of node visits of an optimal run through the graph, that is forced to traverse the known subgraph only on shortest paths. Surprisingly, the performance drops when using macro moves. The reason is the bigger branching factor as explained for the CAU algorithm in Chapter 3.7, but it applies also for conventional UCT.

An optimization used for chance nodes is also implemented in UCT. Each *MOVE* action has a number of successors exponential in the number of edges unknown in the target node, since all of those edge states will be revealed. However, the chance of certain outcomes is very low. It's unlikely, for example, that all edges with a blocking probability smaller than 50% are blocked, and that all those with a high blocking probability are traversable. Still, all child nodes would have to be created if the parent node is expanded. So instead of one chance node, we use a subtree of chance nodes that always reveal only one edge at a time. Each of those chance nodes then has exactly two children, corresponding to one of the edges being traversable or blocked. The effect is that a number of chance nodes equal to the number of edges will be traversed. And instead of one node with an exponential number of children nodes, a linear number of nodes, each with a constant number of children, is visited during the rollout. Unlikely outcomes will then only be created once the number of rollouts becomes really high.

To control the exploration/exploitation dilemma UCT requires a constant C to normalize the estimated cost of the UCT forumla:

$$\overline{X_k} + \underbrace{C\sqrt{\frac{\log n}{n_k}}}_{c(n_k)}$$

This constant should be in the same order of the cost of an average move. Since that move is not known in advance, we use the parent's cost estimate of the node, and multiply it with an exploration parameter. An exploration constant of 1 reflects regular behavior, while lower values lead to a more exploitative algorithm behavior, and larger values lead to more exploration, since the UCT bonus term $c(n_k)$ will be scaled higher. Eyerich, Keller and Helmert obtained their best results with an exploration constant of 0.1.

In previous work [Ald10] we extended the UCT algorithm from Eyerich, Keller and Helmert with remote sensing actions. The results were reasonable for smaller graphs with a high number of rollouts, but rather disappointing for bigger graphs. The increased branching factor due to the large number of available sense actions in each node leads to many complications. Dealing with those problems resulted in a new algorithm derived from UCT which is called CAU and will be presented in Chapter 3.7.

3.7 CAU

A combined approach of all techniques that proved to be successful in solving the CTP with remote sensing is the CTP Adaptded UCT UCT algorithm (CAU). CAU is basically a UCT algorithm with sophisticated search control techniques to handle the higher branching factor. The code of the algorithm we programmed is highly configurable so that it is possible to emulate conventional UCT variants with it. If additional parameters are on, its behavior changes to more sophisticated action selection methods. UCT's anytime-property is conserved, but depending on the chosen parameters, convergence to the optimal solution is not necessarily guaranteed anymore.

3.7.1 General Concepts in CAU

In this section we introduce two general concepts used in CAU: macro moves and a more sophisticated back propagation method to update a node's estimate from its children's cost (see also Chapter 2.3.6).

A requirement for a CTP-solving algorithm is to secure termination. Especially in sampling-based algorithms, loops have to be avoided. The UCT version of Eyerich, Keller and Helmert initializes each node with the Dijkstra distance in the current weather. It is possible that the algorithm alternates between two nodes for a while, usually if a short edge with low blocking probability is blocked, and many of the earlier rollouts improve the estimate of the values in the neighborhood. But since all future rollouts will yield higher estimates, eventually the algorithm will escape the local minimum. When initializing with the help of a table, on the other hand, it is not unlikely that the algorithm gets stuck in a local minimum. After one move, a new successor is generated, that will be initialized with the initial estimate according to a table. In the table the average cost of the node is used, which does not comprise that one of the "essential" edges is blocked. Unlike Dijkstra, a table initialization will then suggest the local minimum again as the best successor, which will be tried out first. Since it is desirable to save runtime and

look up a table value in constant time instead of calculating an above linear estimate, an effort has to be made to ensure termination of each rollout.

A technique that was already used in the UCT of Eyerich, Keller and Helmert is macro moves. The concept is that each *MOVE* must end in a node revealing new information. In Chapter 2.1.3 we have shown that by using macro moves an upper bound for the cost can be given. The "fringe" contains all nodes of the graph that are reachable from the initial state in a known subgraph, but have edges whose status is unknown to the agent yet.

When internal nodes use macro move successors, the algorithm will terminate, since there are at most n nodes in the graph, and each macro move ends in another fringe node. When sensing is allowed it is possible to additionally sense some edges, but still the total number of actions performed during a run is bounded.

Surprisingly, experiments (See Experiment 2, Chapter 4) show that using macro moves does not improve the performance of the algorithm. Whenever the algorithm gets stuck in a blind alley it can immediately choose a good extension among the macro move successors. However, the branching factor gets notably larger because instead of one move back into the known subgraph, all transitive reachable states on the fringe of the graph are possible successors. In the Delaunay graphs we used as benchmark graphs, such dead end roads do not occur frequently enough, compared to the number of rollouts wasted by the increased branching factor.

With the necessity to solve the local maximum problem in CAU, we look for a way to use macro moves but still reduce the branching factor as much as possible. A more sophisticated combination is motivated by observing the algorithm's behavior in many runs. Typically, the agent follows an estimated path along the fringe of the known subgraph, exploring nodes with a decreasing estimated goal distance. Only if some edges that were presumed to be traversable turn out to be blocked, the algorithm has to realign. At this point, macro moves are necessary to avoid flip-flopping. The proposed structure is to use atomic, local *MOVE* child successors, as long as at least one of the edges of the child's target node are unknown. When the statuses of all adjacent edges are already known, the node will receive macro move successors instead. While it is not guaranteed that every *MOVE* action ends in a node that reveals new information, every second move does.

3.7.2 Child Cost Propagation

Another change to the original UCT algorithm is the option to modify the calculation of a node's estimated cost from its children. There are two different methods that can be used to increase the estimate, which can be used

independently or in combination. The first is a different metric for decision nodes as described in Chapter 2.3.6 and the other is a model-based child estimation for chance nodes.

For decision nodes the idea is to use the best cost estimate instead of the accumulated sum of the children's cost, with the reasoning that the agent can decide which action to take. If one child turned out to have an extremely high cost, that might influence the estimate of the parent node for a long time, and it requires a lot of rollouts until the average estimate converges to the real value. This process can be sped up by immediately choosing the cost of the best child as own cost. However, a problem of this method is that initial values have to be encapsulated from those estimates. Otherwise, the algorithm is at risk of propagating initializations that have no base in real rollouts.

Another method is a more informed update of chance nodes. UCT only requires a generative model of its environment but in CTP a precise probability distribution is known to the agent. Not using this information is a waste of available resources. At chance nodes, instead of using the estimates weighted by the number of samples drawn according to the blocking probability of an edge, the agent can use these probabilities immediately.

3.7.3 Initialization

In this chapter we present all modifications made to the basic UCT algorithm. In the experiments section (see Chapter 4) we underline the effect of each of those modifications. However, we do not check for cross-effects. The sum of the benefits of our techniques are not a sum of the benefits of each single modification. It might in fact be possible that some beneficial effects of one of our techniques annihilates or even deteriorates when combined with another technique.

With Dijkstra initialization, every time a node is visited for the first time, it is assumed that there were already n virtual visits before, where the algorithm received the initialization cost. We consider optimistic Dijkstra initialization in three variants: The basic idea ignores the blocking probabilities of the graph, and calculates the distance from the current location to the goal in the optimistic graph. There are two derived modifications (see also Chapter 3.1.2). Instead of using the direct edge costs, it is possible to normalize the costs by the inverse blocking probability. This estimate can then be used either directly (which results in regular Dijkstra with higher path costs) or to select the optimal path. If it is used to select the path only, the path cost will remain original, but which edges are chosen is determined by the normalized cost.

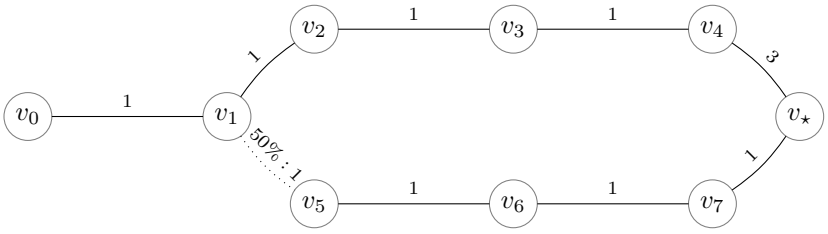

Figure 10: Optimistic Walk-around

The purpose of normalizing Dijkstra estimates is to consider the blocking probabilities and avoid that short but extremely unlikely paths mislead the algorithm. Optimistic walk-around estimation is another approach to estimate the additional cost necessary, if the edge in question is blocked. The detour cost of an edge $e = \{v_i, v_j\}$ is approximated by the smallest distance from its start v_i to its end v_j on an optimistic graph where all other edges are present. This "minimal detour cost" is then multiplied with the blocking probability p, while the actual cost is multiplied by $1 - p$. This sum is then taken as cost estimate for the edge.

Considering an actually possible detour cost of an edge seems like a good idea, even though it is still possible to construct examples, where the estimate of optimistic-walkaround yields bad results. Figure 10 shows an example where it might be better to estimate an edge's cost with another heuristic. Consider the edge $v_1 - v_5$. The cheapest distance from v_1 to v_5 given the edge itself is blocked has a cost of 9 (on the path $v_1 - v_2 - v_3 - v_4 - v_\star - v_7 - v_6 - v_5$. This cost is even higher than the cost of 7 from start to goal. The estimated cost of the edge $v_1 - v_5$ would be $0.5 \cdot 1 + 0.5 \cdot 9 = 5$, so in this figure estimating the edges with optimistic-walkaround will not lead to good results. It is however no surprise that fast calculable heuristic values give just an estimate of the cost that can sometimes be bad. Optimistic walk-around is asymptotically more complex as the Dijkstra variants, because a Dijkstra distance has to be calculated once for every node, which increases the asymptotic complexity from $O(n \log n)$ to $O(n^2 \log n)$.

The core of CAU is a UCT algorithm with a table-based initialization. Using a table instead of recalculating initial Dijkstra estimates changes the runtime of initializing a node from $O(n \log n)$ to $O(1)$. The quality of the algorithm is reduced only by a small margin, while the benefit in faster runtime might outweigh the slightly worse initialization. Given the same runtime, the algorithm's result might even improved, even if we couldn't confirm this in our experiments for reasons described there (see Experiment 3, Chapter 4).

This method also allows more sophisticated initialization techniques like optimistic walk-around that require even more runtime than Dijkstra, since they only have to be calculated once before the algorithm starts. Additionally, there is an option to update the initialization table with the values achieved by actually traversing the algorithm. The table entries represent the aggregated average costs from each node to the goal, ignoring potential information of the roads status. Once the agent is more informed these values do not correspond to the actual costs any more, but they still are a reasonable heuristic. We implemented two methods to reflect that those recent rollouts that are made after the agent has achieved some information are more important than rollouts made in the first run. One method is to discount the table whenever new information is achieved, while the other is to reset the table *visits* to their initial value.

3.7.4 Sensing

The addition of sensing poses several tasks to solve. If sense actions are not treated specially, the algorithm is overwhelmed by the number of options given, which leads to sparse samples with low confidence on each branch. Initially it is hard to determine which sense successors might become beneficial for the search. There are basically two options to deal with the problem: restricting the number of available sense children and modifying their cost estimates. The number of maximum sense children considered at each decision step is a parameter of the algorithm, and heuristics are used to preselect which sense children might be beneficial.

The pseudo-code of the selection of those children is shown here:

```
orderedList SearchNode::getBestSenseChildren(int maximumSenseChildren)
  PriorityQueue childList;
  for all SENSE Children of this node {
    priority=estimatePriorityOf(child)
    if (edge is unknown in current belief) {
      childList.push(child,priority);
    }
  }
  for(i=1 to maximumSenseChildren) {
    orderedList.append(childList.popfont());
  }
  return orderedList
}
```

A priority queue is filled with the estimates of all edges in the graph.

Only the best are then used in this node. There are several ways to estimate the priority of a child. We experimented with:

1. Random: each edge is assigned a priority at random.

2. VoI: Similar to the SAST table used for initializing the nodes, we can store and estimate a value of information for each edge in a table. To initialize this table we simulated an agent following the current initializing policy (e.g. normalized Dijkstra) and compared the cost of a run of the agent given the information of this edge with the cost of an uninformed agent. This difference is computed several times (we used the number of virtual visits of each node) to achieve better results. The normal SAST table contains the estimated distances from each node to the goal, while the "Sense-SAST"-table initially contains the value of information of each edge determined by an (always exploiting) agent simulating that policy. Basically, the Sense SAST initializations are the VoI determined by ORO (with a relatively low number of rollouts).

3. Distance: The priority is the distance of each edge to the current location. Because it is not clear how to compute the distance of an edge to a node, we used the distance to the nearer vertex of the edge plus the cost of the edge itself. The reason behind preferring near edges is that it is unlikely that edges far away in the graph will affect the next move.

4. Dijkstra path: only edges on the Dijkstra path from the current node to the goal are considered. There are two subgroups of this selection method: prioritize edges with higher blocking probabilities or prioritize edges near the current node. This restricts the edges to those considered by Bnaya, Felner and Shimony [BFS09].

The second way to deal with sense children is to modify their cost. Sense fees as described in Chapter 2.5.3 are a way to increase the initial estimate so that the option to sense is only explored once the number of rollouts over the current node is large enough that the UCT bonus term will be as high as the bad initial estimate.

Another way is to compare sense children with the worst instead of the best move child, as presented in Chapter 2.5.3.

4 Experiments

In this chapter we present results achieved with various algorithms. The experiments were made on Delauny graphs since they are a reasonable representation for a street net. Most experiments were run on graphs with 20 to 100 nodes. If different graphs of the same size are tested, they are numbered like graphs 20-0 20-1 and 20-2 in Table 1.

Each graph was tested multiple times for each algorithm, and the average total cost over these runs, including the 95% confidence interval are shown in the tables.

The experiments for this work were run on a 2.7 GHz AMD opteron processor with a memory limit of 3 GB.

4.1 Simulating other Algorithms with CAU

CAU was designed to be a highly configurable algorithm. When the parameters are set accordingly CAU can imitate the behavior of other algorithms. In Table 1 we simulate other algorithms with CAU and compare the results. At first we simulate the Always Sense algorithm (see Chapter 3.1.3). AS senses edges on the Dijkstra path, until the shortest path is found, and then moves along that path. The order in which the edges are sensed matters. The most straightforward implementation starts with sensing the nearest unknown edge on the Dijkstra path. This algorithm can be found in the table under "AS". Bnaya, Felner and Shimony show that, once an intended path is fixed, the best order is to sense edges in order of decreasing blocking probability so that likely blocked roads are sensed first [BFS09]. Our implementation of the AS algorithm, denoted with "AS", does not support this option, however, when using CAU to simulate AS both variants are possible. "CAU AS near" senses the next edge on the Dijkstra path from the current location to the goal and "CAU AS block" senses the edge with the highest blocking probability on that path. To simulate AS on CAU we set the number of rollouts to the maximum branching factor of the graph while the initial visits are set to infinity, so that the algorithm will not converge, but only follows its initial estimates. Sense children are given an incredibly large bonus "fee" (see Chapter 2.5.3) which is a large negative number (fees are intended to punish sensing in early rollouts, so this is an abuse of the original idea). Since the number of rollouts is not high enough to exceed the initialization bonus, sense actions will be chosen if available. To make the algorithm choose exactly the sense child we want, we set the maximum number of sense children to 1.

The available sense children are determined by the top elements of a

Solver	**20-0**	**20-1**	**20-2**	**50**	**100**
AS	200.0±4	214.5±4	146.1±5	228.2±7	478.2±12†
CAU AS near	198.8±4	212.3±4	140.0±5	227.9±7	486.6±49†
CAU AS block	190.7±4	187.9±3	137.1±5	210.9±6	386.9±29†
DIJ	205.3±7	186.5±5	139.5±6	253.2±10	367.2±11†
UCT 10k	167.7±5	148.6±3	131.7±6	185.5±7	286.0±9†
CAU 10k	168.8±6	148.2±3	131.6±5	186.8±7	287.6±8†
UCT 100k	167.7±5	148.7±3	132.5±6	185.6±7	286,0±9†
CAU 100k	167.9±5	148.9±3	132.1±6	185.5±7	–†

Table 1: Algorithms simulated on CAU; †: based on $<$ 1000 runs

priority queue (see Chapter 3.7.4). The parameters of CAU allow to select the nearest edges on the Dijkstra path or the edge on the Dijkstra path with the highest blocking probability. These options match exactly to the two heuristics of the AS-algorithm. In Table 1 the experimental results approve that CAU can behave similar to AS. The slightly better performance of CAU, which becomes especially apparent on graph 20-2 is explainable to a hard-coded improvement: when a node has only one *MOVE*-child, the next action will be chosen as moving to that node. There, information of adjacent edges gets revealed for free that a regular AS algorithm would have to sense.

The next algorithm we simulated with CAU is the UCT variant of Eyerich, Keller and Helmert [EKH10]. The result we obtained was slightly better than theirs, due to a minor bug in their code. The best configuration of UCT uses macro moves only in root nodes. In the CAU algorithm such an option is not available, since it might lead to non-terminating runs, when initialized with the SAST-table. However, in CAU exists the more sophisticated "use macro moves only if needed" option described in Chapter 3.7.1 which yields similar results, as can be seen in Table 1. The Eyerich UCT algorithm was tested with 10 000 rollouts (UCT 10k) and 100 000 rollouts (UCT 100k), while the CAU simulating UCT is found under CAU 10k and CAU 100k. The difference in performance is small, and it can also be seen that the algorithm already converged to a rather stable result, since the difference between 10 000 and 100 000 rollouts is small, and on some graphs more rollouts even yield slightly worse results. Nevertheless, the difference between all four configurations is not significant. For the graph with 100 nodes, some runs exceeded the memory limit, and it generally took a long time to finish. instead of 1000 only half the runs were completed, when we chose to prematurely stop the experiments. The high variance indicates this lack of rollouts, and the experimental results should be treated with care.

Solver	20-0	20-1	20-2
CAU everywhere	170.3±8	†	†
CAU when needed	169,6±8	†	†
UCT everywhere	171.4±6	150.7±3	133.2±6
UCT root	168.1±5	149.0±3	132.5±6

Table 2: Macro Moves increase the branching factor; †: no experiments

4.2 Macro Moves

In Table 2 we show that using macro moves in every node "CAU/UCT everywhere" is not as good as using macro moves only when needed (see Chapter 3.7.1) "CAU when needed" or only in root "UCT root", which are the respective options in UCT and CAU. Even though the result for CAU in the table is not significant, it does not mean that the effect of macro moves was negligible in general. We manually investigated the average branching factor for CAU in a number of experiments and realized that it was always higher in the graphs considered. The impact of using the macro moves only when needed is not huge, but significant. We did not find the time to compare "macro moves only when needed" from CAU and "macro moves only in root" which is only applicable with a Dijkstra initialization but not with a SAST table.

4.3 SAST table

Experiment table 3 compares different Initialization methods. The results are based on 300 runs per entry. On the smaller graphs the influence of using a table initialization is rather small and even if it becomes greater on larger graphs, the difference is still not significant. When comparing the runtime of the different approaches the Dijkstra initialization took around 3000 seconds on average on the graphs with 20 nodes while the table initialized approaches used approximately twice the time. The effect that Initializing nodes is reduced from an asymptotic $O(n \log n)$ to $O(1)$ is unfortunately not underlined by this experiment here. Sadly former experiments, underlining a significant gain in runtime were not conserved. CAU is not optimized for speed yet and a parameter in the manner of resetting the table values might have led to unnecessary extra calculations.

Solver	20-0	20-1	20-2	50
CAU 10k Dij	172.2±10	152.3±6	137.1±10	184.8±12
CAU 10k SAST	172.6±11	152.3±6	138.0±10	188.2±11
CAU 12k SAST	171.8±10	152.0±6	139.8±10	191.5±13
CAU 15k SAST	171.9±10	152.5±6	137.4±10	192.4±12

Table 3: Using a SAST table initialization does not deteriorate the results significantly

Solver	20-0	20-1	20-2	50
CAU UCT	168.8±6	148.2±3	131.6±5	186.8±7
CAU bad propagation	202.0±9	170.0±5	159.4±10	285.1±17

Table 4: Using the best child in decision nodes and children according to model in chance nodes

4.4 Child Cost Propagation

A combination of the approaches of Chapter 2.3.6 and Chapter 2.3.4 leads to a different way of updating the parents cost from its children. An issue here is that UCT has to be initialized to perform properly. However, it has to be secured that those initial values only influence the child node and are not propagated to its parent. In Table 4 we see the effect of initial values getting propagated, "CAU bad propagation", compared to the regular UCT update, "CAU UCT".

For all experiments of Table 4 we used 10 000 rollouts. The values in this table rely on 500 runs for each node.

4.5 Punish Sensing

In Experiment 5 we

can see that the state space becomes to large to find a solution that is as good as UCT without sensing when allowing to sense all edges at each point in time. By restricting the number of possible sense actions and only allowing those that deem beneficial by a heuristic – in this case a simulated agent similar to ORO – the performance stays approximately constant for a different number of maximum sense children. Still the problem remains that sensing will be chosen too often. So avoid this the trick described in Chapter 2.5.3 is used, and sense children are estimated with a difference from best to worst child, if the estimated benefit of sensing the edge is smaller than its sensing cost. With this policy, the gain from choosing beneficial sensing equals approximately the cost of useless sense actions, at least for up to 5

Solver	20-0	20-1	20-2	50
CAU no sense	167.5±5	148.7±3	132.4±6	184.9±7
CAU 1 edge wc	167.7±5	148.6±3	132.1±6	184.6±7
CAU 2 wc	167.8±5	148.2±3	132.9±6	186.9±7
CAU 3 wc	166.8±5	148.0±3	132.8±6	202.9±9
CAU 5 wc	167.4±5	148.1±3	132.3±6	238.5±12
CAU all edges wc	205.5±5	186.5±5	143.7±6	503.0±8

Table 5: Sensing with a limited number of maximum sense children

Solver	20-0
CAU RAVE	167.9±5
CAU RAVE + CCP	183.8±6

Table 6: Initializing with RAVE

sense children at each decision step on the smaller graphs. Once all edges in the graph are sensable "CAU all edges" the result becomes worse. On the larger graph with 50 nodes it becomes apparent that the bad effect from careless sensing occurs with a lower number of sense children.

wc: worst compare

4.6 RAVE

The results obtained when experimenting with different RAVE tables suggest the occurrence of a bug. The result of the algorithm was independent from the number of initializations put into the RAVE table. The child cost propagation has problems when initial values are propagated as cost estimates for other nodes. In our experiment using child cost propagation led to worse results which was not expected for RAVE.

To get reliable results takes multiple hours and rather weeks. Due to the large amount of configurations it sometimes is not easy to know how to set the parameters up front. Unfortunately, we did not have enough time to find good parameters for every configuration.

5 Conclusion and Future Work

In this work we analyzed algorithms for the Canadian Traveler's Problem with remote sensing. Sampling algorithms like UCT are one of the few tools available to cope with the large search space. However, the samples must be guided to converge to a good solution in reasonable time. We investigated different search control techniques.

5.1 Future Work

As of now, the search nodes are stored in a tree structure. With the availability of remote sensing there is a relatively high number of nodes in that tree that share the same belief. Those nodes could be unified to have more rollouts and as such a more informed estimate of those nodes. If nodes need to be deleted once in a while to save memory, it is even thinkable to implement a counter, indicating the number of parents a given node has. The total number of possible belief states is restricted to $3^{|e|} \cdot |n| \cdot 2$ where $|e|$ is the number of edges in the graph and $|n|$ the number of nodes. The belief of each edge can be either `unknown`, `traversable` or `blocked`, the agent can be located at $|n|$ different nodes and with the option to have either direct *MOVE*-children or macro move successors there are potentially two different types of nodes for each belief state.

Another idea is a carrying on of the ideas presented in Chapter 2.3.6 and 2.3.4. The UCT algorithm is an extension of UCB, a bandit algorithm. It seems unlikely that another algorithm may perform significantly better on problems, where a generative model is available. However, in the case of CTP we do not only have a generative Model but we have a complete world description as we know all edge blocking probabilities exactly. This information is not taken into account, and therefore a logarithmic overhead is generated. However, recall the UCB idea (see Chapter 2.3.2) of a multi-armed bandit. If the agent already knows the exact probabilities of each arm, it is not necessary to try out suboptimal arms at all. The best policy is to always pull the arm with the highest reward. In the CTP it is still not feasible to exactly calculate the costs of each arm, but also here a sampling based approach can be used. Instead of basing exploration and exploitation on a formula minimizing the regret, the amount of coverage of the search space could be used to guide the rollouts.

One of the initialization heuristics, optimistic walk-around (see Chapter 2.3.5), has the potential to generate more realistic estimates, but suffers from a relatively high runtime, a drawback which could be overcome by using a table based approach like the RAVE tables presented in Chapter 2.5.2.

5.2 Conclusions

The CTP with remote sensing is a hard problem, and not only for computers, but also for humans it is difficult to determine if sensing an edge is beneficial or not. Some of the ideas proposed here did not have the expected impact. With the actions being non-deterministic, judging an algorithm's behavior becomes difficult as its variance is typically quite large. Many samples are required to see the impact of even minor modifications.

Searching for an algorithm that is able to sense for a beneficial effect discovered many interesting nuances of the CTP. Unfortunately, many of these nuances caused problems, but in the end we found solutions for most of them, combined in a new UCT solver, CAU. Finding a clever set of parameter settings should enable us to produce solutions of higher quality in the future. The techniques presented in this work are generally formulated. Therefore, they are suited to not only tackle the CTP but many other problems sharing some of its characteristics, especially other instances of deterministic POMDPs.

References

[ACBF02] Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. Finite-time Analysis of the Multiarmed Bandit Problem. In *Machine Learning 74*, pages 235–256, 2002.

[Ald10] Johannes Aldinger. Approaching the Canadian Traveler's Problem with Remote Sensing, 2010.

[BFS09] Zahy Bnaya, Ariel Felner, and Solomon E. Shimony. Canadian Traveler Problem with Remote Sensing. In *Proceedings of the 21st IJCAI Conference*, pages 437–442, 2009.

[Dij59] Edsger W. Dijkstra. A Note on Two Problems in Connexion with Graphs. In *Numerische Mathematik 1*, pages 269–271, 1959.

[EKH10] Patrick Eyerich, Thomas Keller, and Malte Helmert. High-Quality Policies for the Canadian Traveler's Problem. In *Proceedings of the 24th AAAI Conference*, pages 51–58, 2010.

[FB11] Hilmar Finnsson and Yngvi Björnsson. Cadiaplayer: Search-control techniques. *KI*, pages 9–16, 2011.

[GS07] Sylvain Gelly and David Silver. Combining Online and Offline Knowledge in uct. In *Proceedings of the 24th ICML Conference*, pages 273–280, 2007.

[KMN99] Michael Kearns, Yishay Mansour, and Andrew Y. Ng. A Sparse Sampling Algorithm for near-optimal Planning in Large Markov Decision Processes. In *Proceedings of the 16th IJCAI Conference*, pages 1324–1331, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[KS06] Levente Kocsis and Csaba Szepesvári. Bandit based Monte-Carlo Planning. In *Proceedings of the 6th ECML Conference*, pages 282–293, 2006.

[Lit96] Michael Lederman Littman. *Algorithms for Sequential Decision Making*. PhD thesis, Brown University, 1996.

[NK08] Evdokia Nikolova and David R. Karger. Route Planning under Uncertainty: The Canadian Traveller Problem. In *Proceedings of the 22nd AAAI Conference*, pages 969–974, 2008.

[PY91] Christos H. Papadimitriou and Mihalis Yannakakis. Shortest Paths without a Map. In *Theoretical Computer Science 84*, pages 127–150, 1991.