REYKJAVÍK UNIVERSITY
HÁSKÓLINN Í REYKJAVÍK

Solving General Game Playing Puzzles using Heuristic Search

Gylfi Þór Guðmundsson

Master of Science
June 2009

Reykjavík University - School of Computer Science

M.Sc. Project Report

Solving General Game Playing Puzzles using Heuristic Search

by

Gylfi Þór Guðmundsson

Project report submitted to the School of Computer Science
at Reykjavík University in partial fulfillment
of the requirements for the degree of
Master of Science

June 2009

Project Report Committee:

Dr. Yngvi Björnsson, supervisor
Associate Professor, Reykjavík University

Dr. Ari Kristinn Jónsson
Dean of School of Computer Science, Reykjavík University

Dr. Björn Þór Jónsson
Associate Professor, Reykjavík University

Solving General Game Playing Puzzles using Heuristic Search

by

Gylfi Þór Guðmundsson

June 2009

Abstract

One of the challenges of General Game Playing (GGP) is to effectively solve puzzles. Solving puzzles is more similar to planning algorithms than the search methods used for two- or multi-player games. General problem solving has been a topic addressed by the planning community for years. In this thesis we adapt heuristic search methods for automated planning to use in solving single-agent GGP puzzles.

One of the main differences between planning and GGP is the real-time nature of GGP competitions. The backbone of our puzzle solver is a real-time variant of the classical A* search algorithm we call *Time-Bounded and Injection-based A** (TBIA*). The TBIA* is a complete algorithm which always maintains a best known path to follow and updates this path with new and better paths as they are discovered.

The heuristic TBIA* uses is constructed automatically for each puzzle being solved, and is based on techniques used in the Heuristic Search Planner system. It is composed of two parts: the first is a distance estimate derived from solving a relaxed problem and the second is a penalty for every unachieved sub-goal. The heuristic is inadmissible when the penalty is added but typically more informative. We also present a caching mechanism to enhance the heuristic performance and a self regulating method we call $adaptive\ k$ that balances cache useage.

We show that our method both adds to the flora of GGP puzzles solvable under real-time settings and outperforms existing simulation-based solution methods on a number of puzzles.

General Game Playing þrautir leystar með upplýstum leitaraðferðum

eftir

Gylfi Þór Guðmundsson

Júní 2009

Útdráttur

Eitt af viðfangsefnum alhliða leikjaspilara er að fást við einmenningsleiki eða þrautir. Að leysa slíkar þrautir er mjög ólíkt því að spila gegn andstæðingum og á meiri samleið með reikniritum fyrir áætlunargerð. Í þessari ritgerð er byggt á margra ára rannsóknum á almennum áætlunarreikniritum og þær aðferðir heimfærðar yfir í heim alhliða leikjaspilara.

Meginmunurinn á áætlanagerð og alhliða leikjaspilun er að leikjaspilunin er háð tímatakmörkunum þar sem leikmenn fá upphafs- og leikklukku. Kjarninn í lausnaraðferð okkar er rauntíma útfærsla af A* leitaraðferðinni sem við köllum *Time-Bounded and Injection-based A**.

Stöðumatið sem við notum byggir á hugmyndum frá Heuristic Search Planner áætlunar hugbúnaðinum og er tvíþætt. Annars vegar er vegalengdin í mark áætluð með því að leysa einfaldaða útgáfu af vandamálinu og hins vegar er bætt við refsingu fyrir hvert óuppfyllt lausnarskilyrði. Vegna þess að ein aðgerð getur uppfyllt fleiri en eitt lausnarskilyrði er ekki tryggt að stöðumatið okkar sé lágmarkandi en í mörgum tilfellum er það mun nær raunveruleikanum sem aftur flýtir fyrir leitinni. Þar sem stöðumatið er tímafrekt kynnum við uppflettiaðferð sem flýtir fyrir útreikningi stöðumata. Einnig höfum við sjálfstillandi ávörðunartöku sem við köllum $adaptive\ k$ sem nýtir sér uppflettingar eftir gæðum þeirra.

Við sýnum fram á að fyrrgreindar aðferðir virka vel á fjölda þeirra þrauta sem notaðar hafa verið í alþjóðlegum keppnum og að við höfum bætt við þann fjölda þrauta hægt er að leysa.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

With the development of the computer in 1941 the technology to create intelligent machines finally became available. In the early days the expectations where high and everything was going to be solved with the newly developed and powerful computing machines. In 1950 Alan M. Turing introduced the ”Turing Test”, that would prove the intelligence of the machines, and Claude Shannon was analyzing chess playing as a search problem, in the belief that if the computer could beat a human in chess it surely must be considered intelligent.

The ”Turing Test” stands unbeaten but computers have mastered the art of playing chess. IBM's Deep Blue became a household name in 1997 when it beat chess world champion Garry Kasparov in a six game chess match. Deep Blue as well as today's game-playing programs are very specialized with domain specific knowledge embedded into their program code. Such programs have therefore no premises to deal with other problem domains. Simply put, if presented with the simple game of Tic Tac Toe, Deep Blue would not even know where to begin.

A general solver must be able to solve new and unseen problems without human intervention. The planning community has held International Planning Competitions (IPC) for general planners since 1998. The competitions have been a great success in that they provide a common testing ground, standardize problem representation and provide a yard stick for developers to measure progress. In light of the success of these competitions, the logic research group at Stanford University started the annual General Game Playing (GGP) competition. In the spirit of general planners, GGP systems are capable of playing many different games without the need for pre-coded domain knowledge. The main difference between planning and GGP, however, is that GGP is a real-time process where players have two time constraints and must perform legal actions on a regular basis. The

times allocated before first action are typically 20-200 seconds and then 10-30 seconds for every move there after.

CADIA-Player (Finnsson, 2007) is a competitor from Reykjavik University and has participated in the GGP competition since 2007. It is the current and two time GGP champion, winning the 2007 and 2008 competitions. For games with multiple players a Monte Carlo simulation-based approach called UCT (Kocsis & Szepesvári, 2006) has proved quite successful. However, the 2007 GGP competition showed that for many single-player games the simulation-based approach suffers from scarce feedback, i.e. goal scoring states are rare and hard to find.

In this thesis we present the work done to enhance CADIA-Player's ability to deal with single-player games prior to the 2008 competition. The main contributions are a new search method we call *Time-Bounded and Injection-based A** (TBIA*), that is inspired by Time-Bounded A* (Björnsson, Bulitko, & Sturtevant, 2009) and a heuristic function that solves a relaxed problem, adapted from the planning community. As the heuristic is the main bottleneck of our system we propose a caching mechanism for the heuristic and a self regulating algorithm we call $adaptive\ k$ that balances the use of the cache according to its quality.

We start in Chapter 2 by analyzing problem definition languages as well as landmark ideas that made informed search methods the mainstay of general planners. We then move on to Chapter 3 where we describe the TBIA* search algorithm we use in our implementation. Chapter 4 shows how we derive a heuristic for the informed search method as well as discussing the enhancements we made. We validate our method with empirical results from several games in Chapter 5 and finally we conclude this paper with a summary and conclusions in Chapter 6.

Chapter 2

Background

In domain-independent planning the planner does not know the problem a priori and can thus not rely on domain specific information to solve the problem at hand. Any information to guide the solver must be automatically extracted.

The problems are described using a formal language. The first part of this chapter is devoted to such languages. We start by describing the languages used by the planning community. First are the STRIPS (Fikes & Nilsson, 1971) and PDDL (Bacchaus, 2001; McDermott, 1997) languages and then we describe GDL (Love, Genesereth, & Hinrichs, 2006) that is used in GGP.

The second part of this chapter is devoted to the ideas and implementations of planning systems and how they automatically derive search guidance heuristics from the problem description. We are particularly interested in state space based planners as they use search techniques that also apply to GGP. We also briefly describe a few other ideas that have successfully been applied to derive heuristics.

2.1 Representation of Problem Domains

A high level abstraction of a planning problem is the challenge of discovering a path from one state to some other more favorable state (goal state) by applying operators (state transitions) available at each intermediate state traversed on the way toward the favorable state. A state is defined in first order logic where the smallest unit of knowledge is an atom. Each state is then comprised of an atom set with either a true or false value. The set A is the superset of all possible atoms within the problem domain (every possible bit of knowledge knowable within the domain). The operators of the domain are all possible

transitions from some one subset of atoms in A to some other subset of atoms in A, i.e. operators can be split into three categories:

- Prec(o) is the set of atoms required to be true for the operator to be applicable in the current state.
- Add(o) is the set of atoms and value added to the next state by applying the operator.
- Del(o) is the set of atoms and values removed in the next state.

The need to unify and standardize the problem description was apparent and following is a brief description and history of the languages used in planning.

2.1.1 STRIPS

The STRIPS planning problem modeling language was derived from one of the oldest planning systems (Fikes & Nilsson, 1971) and the three key assumptions it made:

1. That the planner has complete information about all relevant aspects of the initial world state.
2. That the planner has a perfectly correct and deterministic model of the effects of any action it can take.
3. That no change in the world is ever caused by anything other than the actions made by the planner.

Even under these harsh restrictions the planning problem is computationally hard. The problem P is represented as a tuple, $P = \langle A, O, I, G \rangle$ where:

- A is the set of all possible atoms,
- O is a set of all ground operators where each operator is a tuple $\langle Prec, Add, Del \rangle$ where:
  - $Prec$ is a set of atoms that must be true in current state for this action to be applicable,
  - Add is a set of atoms that become true by applying the operator,
  - Del is a set of atoms that become false by applying the operator.
- $I \subseteq A$ is the subset of atoms representing the initial state,
- $G \subseteq A$ is the subset of necessary atoms for a state to be a goal state.

The state-space determined by the problem P is a tuple $S = \langle S, s_0, S_G, A(.), f, c \rangle$ where:

1. The states $s \in S$ are collections of atoms from A and S is the set of all possible states.
2. The initial state s_0 is defined by a set of atoms in I.
3. The goal states $s \in S_G$ are all states such that $G \subseteq s$, i.e. if the state s contains all the atoms in the goal conditions set G it is a goal state.
4. The action $a \in A(s)$ is the operator $o \in O$ such that $Prec(o) \subseteq s$, i.e. $A(s)$ is a set of all available actions(specific operator) in the current state s.
5. The transition function f maps states s to states $s' = s - Del(a) + Add(a)$ for $a \in A(s)$.
6. The action costs c(a) are assumed to be 1.

STRIPS uses grounded atom sets, i.e. no formulae, for initialization, goal conditions and preconditions of operators. This simple STRIPS language is still used, often for demonstration purposes or as a stepping stone to the more complicated representations. In 1988 a richer modeling language was introduced, Action Description Language (ADL) (Pednault, 1989). In ADL actions are allowed to have more complicated preconditions (essentially first-order formulae involving negation, disjunction and qualification) and effects that may depend on the state in which the action is taken.

2.1.2 Planning Domain Definition Language

With the first international planning competition (IPC-1) the problem description languages were standardized and called Planning Domain Definition Languages or PDDL (Bacchaus, 2001; McDermott, 1997). In Figure 2.1 we can see parts of the definition for sliding tile puzzles as well as one instantiation of the 3x3 version called 8-puzzle. The aim of standardizing the languages was to simplify and encourage problem sharing among researchers, making comparison of results easer and making the competition at AIPS-98 possible. There have been several alterations made to the PDDL standard since IPC-1. PDDL 2.1 (Long & Fox, 2003) used at IPC-2 in 2003 was the most significant change and the latest version, PDDL 3.0, was used at IPC-6 in 2008. The most significant changes are:

1. PDDL inherited features from the Action Description Language (ADL) where first order formulae are allowed as preconditions for actions as well as action effects depending on the state they are taken in.

Definition file:

```
(define (domain strips-sliding-tile)
  (:requirements :strips)
    (:predicates
      (tile ?x) (position ?x)
      (at ?t ?x ?y) (blank ?x ?y)
      (inc ?p ?pp) (dec ?p ?pp))
    (:action move-up
      :parameters (?t ?px ?py ?by)
      :precondition (and
        (tile ?t) (position ?px) (position ?py) (position ?by)
        (dec ?by ?py) (blank ?px ?by) (at ?t ?px ?py))
      :effect (and (not (blank ?px ?by)) (not (at ?t ?px ?py))
        (blank?px?py) (at ?t ?px ?by)))
      ...
```

Puzzle file:

```
(define (problem hard1)
  (:domain strips-sliding-tile)
  (:objects t1 t2 t3 t4 t5 t6 t7 t8 p1 p2 p3)
  (:init
    (tile t1) ... (tile t8)
    (position p1) (position p2) (position p3)
    (inc p1 p2) (inc p2 p3) (dec p3 p2) (dec p2 p1)
    (blank p1 p1) (at t1 p2 p1) (at t2 p3 p1) (at t3 p1 p2)
    (at t4 p2 p2) (at t5 p3 p2) (at t6 p1 p3) (at t7 p2 p3)
    (at t8 p3 p3))
  (:goal
    (and (at t8 p1 p1) (at t7 p2 p1) (at t6 p3 p1)
    (at t4 p2 p2) (at t1 p3 p2)
    (at t2 p1 p3) (at t5 p2 p3) (at t3 p3 p3)))
)
```

Figure 2.1: Parts of the PDDL description for 8-puzzle

2. The use of real-valued functions in the world model, and actions whose preconditions include inequalities between expressions involving those functions. This makes the world model essentially infinite, and therefore it is possible to specify undecidable problems in PDDL 2.1 (Helmert, 2003).

3. The possibility to specify the duration of actions for temporal planning and scheduling.

4. The possibility to specify different kinds of plan metrics for optimal planning.

5. Derived predicates, predicates that are not affected by any of the actions available to the planner. Same as "axioms" in original PDDL but never previously used in competition.

6. Timed initial literals which are a syntactically very simple way of expressing a certain restricted form of exogenous events: facts that will become TRUE or FALSE at time points that are unknown to the planner in advance, independently of the actions the planner chooses to execute.

7. Soft goals, or valid goals that a valid plan does not have to necessarily achieve.

8. State trajectory constraints, which are constraints on the structure of the plans and can be either hard or soft. Hard trajectory constraints can be used to express control knowledge or restrictions on the valid plans in a planning domain and soft trajectory constraints can be used to express preference that affect the plan quality, without restricting the set of valid plans.

2.1.3 Game Description Language

GGP uses the Game Description Language (GDL) (Love et al., 2006) as the standard to describe the rules of a game. GDL is variant of *Datalog* which is a query and rule language similar to *Prolog* and the description files use the Knowledge Interchange Format (KIF). GDL allows for single or multi-agent games and the games can be adversary and/or co-operative in any combination. Multiple goal conditions are allowed ranging in value from 0-100 and multi agent games are not necessarily zero-sum games. The two main restrictions on GDLs expressiveness are that game descriptions have to be deterministic and provide complete information. There are 8 keywords that cover the state machine model of any GDL game: *role*, *init*, *true*, *does*, *next*, *legal*, *goal*, and *terminal*. Following are descriptions of the relations and a brief example.

```
(role player)
(init (cell 1 1 A))
(init (cell 1 2 b))
...
(init (step 0))
(<=   (legal player (move ?x ?y))
      (true (cell ?u ?y b))
      (or (succ ?x ?u) (pred ?x ?u)))
...
(<=   (next (cell ?x ?y b))
      (does player (move ?x ?y)))
...
(succ 1 2)
(succ 2 3)
(<=   (goal player 100)
      inorder)
(<=   (goal player0)
      not inorder)
(<=   (terminal inorder))
```

Figure 2.2: Parts of a simplified GDL for 8-Puzzle

In Figure 2.2 we have a few selected lines from the 8-puzzle GDL file. A game description starts with the declaration of the roles of the game, this is done with the *role* keyword and once declared the roles of the game can not change. The second step is defining the initial state. This is done using the *init* keyword that takes a formula or atom as input and asserts them as facts in the initial state. The *true* keyword works in a similar way but validates if the given atom or formulae hold in the current state.

To define actions the *legal* keyword is used. *Legal* takes a role as parameter and its solutions are the legal actions for that role in the current state. As can bee seen in Figure 2.2 a *legal* defines the action *move* for the *role* player and *move* takes two parameters $?x$ and $?y$. Any atom beginning with "?" is a variable in KIF and will be replaced with any valid value available from the current state. The second line of the *legal* definition, $(true\ (cell\ ?u\ ?y\ b))$, enforces a precondition required for the $move$ action to be available, namely that the adjacent cell contains the value "b" indicating it is blank. Once the agent has chosen an action to perform it is wrapped up with the role name in a *does* relation and asserted into the state. Note that in a multi *role* game actions are performed simultaneously for each player every turn. In turn taking games, such as Chess, the turn taking is simulated by forcing players to choose a no-op action every other turn.

The state transitions are done via relations defined by the *next* keyword. As is shown in Figure 2.2 the example *next* relation has a precondition of $does\ player\ (move\ ?x\ ?y)$ so

it adds atoms to the new state as a result of a *move* action being performed. The new atom is the fact that some $cell\ ?x\ ?y$ moved from should contain the blank tile in the next state. Unlike PDDL it is not just the effects of actions that are propagated but every atom that should still hold has to be propagated to the new state by some *next* relation. Some facts such as the *does* and any delete effects of the action chosen are simply not propagated. In essence this means that the state transition in GDL requires much more work as it is not only dependant on the additive effect of a action but also the size of the state. How the state transitions are performed is one of the biggest differences between PDDL and GDL.

The *goal* relations take two parameters, role and score, where the score can be any value from 0-100. There can be one or more atom and/or formulae defining the precondition for that goal to hold for the given role. In the simplified 8-puzzle GDL example both goal condition have a formulae referencing a function $inorder$ that is not include in the example. The $inorder$ function is simply a list of *true* relations placing tiles in the correct cells similar to the *init* list in the example.

The game will continue until a terminal relation is satisfied, defined using *terminal* and some set of atoms.

Notice that in GDL goal-, terminal-, initial- and legal relations allow first order formulae including negations.

GDL does not support real-valued functions other than for goal values and hence the use of the $succ$ relation in the example in Figure 2.2, third line of the $legal$ relation. The $succ$ relation is basically defining increment by one.

For the complete specification of GDL see *"General Game Playing: Game description language specification"*(Love et al., 2006).

2.2 Search and Planning

At the time of IPC-1 in 1998 the state-of-the-art planners where based on Graphplan (Blum & Furst, 1995). Graphplan systems build graphs where states are nodes and actions are edges. To prune this graph the method keeps track of contradicting atoms, mutex relations, and prunes states accordingly. There was one entry in IPC-1 that was based on heuristic search, Heuristic Search Planner (HSP) (Bonet & Geffner, 2001), and did well enough to change the field. IPC-2 was held in 2000 and by then the heuristic planners

dramatically outperformed the other approaches with regard to runtime. This caused the trend towards heuristic planners to increase still.

2.2.1 Heuristic Search Planner, HSP and HSPr

The first search based planner was Heuristic Search Planner (Bonet & Geffner, 2001) (HSP or HSP1) which participated in the International Planning Competition, IPC-1, in 1998. HSP was a non-optimal planner that applied informed search methods to solve the planning problems. Informed search methods are better than brute force search only if they have a good heuristic to guide the search. What makes a heuristic good is its ability to evaluate the current state. In other words it can provide, with less effort than doing the search, an estimate of how far a given state is from a goal state. There are two definitions that we need to keep in mind.

- Informativeness is how close the heuristic estimate is to the true value.
- Admissibility is when there is never an overestimate of the true value.

If the problem is solvable at all, an admissible heuristic and a complete search method guarantee the discovery of a optimal solution. In HSP the algorithm was not optimal because the greedy search method was not complete and the heuristic was not admissible. However, the heuristic was quite informative which made the system find relatively good solutions quite fast. The distinction between optimal and approximate solutions is important as they often fall into different complexity classes (optimality may be much harder). In the planning community, emphasis on optimality is not always an issue. It can be more important to find a valid solution fast rather than having a optimal solution that took much longer to acquire. HSPs implementation can be summarized by:

- It works on the STRIPS problem description language.
- The search is progression based and has to compute the heuristic value in every state.
- The heuristics $h(s)$ is derived as an approximation of the optimal cost function of a "relaxed" problem P in which the delete lists of operators are ignored.

The relaxation works as follows. The first step is to give every proposition p a value $v(p)$ of either zero if it is part of the initial state or infinite otherwise. Then the approximate search tree is grown as follows; for every operator op available in the current state s, add

all propositions in the operators add lists, $Add(op)$, with value of

$$v(p) = \min\left[v(p), 1 + v(Prec(op))\right]$$

where $Prec(op)$ is a list of propositions that made the operator possible[1]. This process halts when the values of propositions, $v(p)$, do not change. In essence what happens is that each step merges another level of the search tree onto the growing super state. When this process halts, the $v(p)$ value assigned to each proposition is a lower bound estimate on the cost of achieving it from the initial state. Either all propositions required for the goal conditions have been estimated by a value less then infinite or the problem is not solvable from the current state. By assuming that goals are fully dependent, a heuristic guaranteed to be admissible can be derived by using the highest valued proposition required to satisfy the goal condition,

$$h(s) = \max_{p \subseteq G}\left[v(p)\right]$$

where G is the set of goal propositions. This estimate, however, may be far lower than the true value, i.e., uninformative. If the heuristic is uninformative the search will most certainly need to explore more states and the search progress will be slow, this is also known as thrashing. For this reason HSP chooses to assume that the sub-goals are independent, i.e., achieving them has no positive interactions, thus making it safe to sum up all the estimated values of the goal condition,

$$h(s) = \sum_{p \subseteq G} v(p)$$

where G is the set of goal propositions. This heuristic estimate is much more informative. But as the assumption of sub-goal independence is not true in general, this estimate is not admissible.

The main bottleneck in HSP is the frequent heuristic calculations (taking more than 80% of the time). To counter this problem HSP used a form of hill-climbing search method that needs fewer heuristic derivations but often finds poor solutions. Another version, HSP2, uses a weighted A* (WA*) where high weight value results in faster search but poor solutions. In the IPC-1 competition HSP did surprisingly well, solving 20% more problems than the GraphPlan and SAT based planners (both optimal approaches) but for many of the problems HSP had poor solutions (Hoffman, 2005).

HSPr is a variation of HSP that removes the need to recompute the proposition costs $v(p)$ for every state. This is achieved by computing these costs once from the initial state

[1] If there are more then one precondition, the one with the highest estimate should be used.

and then performing a regression (backward) search from the goal to the start state. The heuristic estimate $h(s)$ for a given state s is computed from s_0 as

$$h(s) = \sum_{p \subseteq s} v(p)$$

The main premise for this process is defining the *regression space* that is used for the regression search. The *regression space* is an analogy to the *progression space*[2] where:

- the states s are sets of propositions (atoms) from A, the set of all propositions,
- the initial state s_0 is the goal G,
- the goal states $s \in S_G$ are the states for which $s \subseteq I$ holds,
- the set of actions $A(s)$ applicable in s are operators $op \in O$ that are *relevant* and *consistent*; namely, for which $Add(op) \cap s \neq 0$ and $Del(op) \cap s = 0$,
- the state $s' = f(a, s)$ that follows the application of $a \in A(s)$ is such that $s' = s - Add(a) + Prec(a)$.

A solution in the *regression space* is the inverse of the solution in the *progression space*, but the forward and backward search spaces are not symmetric. The state $s = \{p, q, r\}$ in the regression space stands for the set of states s where $\{p, q, r\} \subseteq s$ in the *progression space*. The proposition estimates $v(p)$ are then calculated as before using the *regression space*. The advantage is that once this is done, the values can be stored in memory and looked up as there is no need to recalculate them. With the lower cost of obtaining the heuristic estimate a more systematic search algorithm is feasible giving better solutions. The regression search, however, can generate states that violate basic invariants of the domain, i.e., there can be some state s_r in the *regression space* which has no superset s_p. To prevent spending effort on such cases HSPr identifies proposition pairs that are unreachable from the initial state (a temporal mutex) and prunes such states from the search.

2.2.2 Other Planning Systems

Some of the strongest planners at the time of HSPs debut were based on GraphPlan (Blum & Furst, 1995; Kambhampati, Parker, & Lambrecht, 1997). Instead of applying search on the problem domain itself GraphPlan creates a "unioned planning-graph" of the forward state-space search tree. The graph is a directed acyclic graph (DAG) where each node

[2] See section 2.1.1 about STRIPS for better reference.

(super state) is a compact disjunctive representation of all states on each level of the search tree and edges are operators that map propositions from one node to the next. To prevent exponential blowup of edges in the graph, actions are now validated against the super state, the algorithm keeps track of all 2-sized proposition subsets that do not belong to legal states. Solution extraction is then performed with a recursive backtracking search from the last node to the initial node, looking for a partial solution path. At IPC-2 in 2000 the ideas of GraphPlan made a comeback in the Fast Forward (FF) Planning system (Hoffman & Nebel, 2001) by winning the award for "distinguished performance planning system". FF used GraphPlan as its heuristic by having it solve relaxed puzzles and useing the derived solutions length as a lower-bound estimate to the true solution length. Using GraphPlan as the heuristic is feasible as the relaxed puzzle does not contain any of the expensive mutex relations of the original problem and a solution can be extracted in a single backtracking sweep. GraphPlan's main advantage over HSP is that it does not lose information over proposition dependency within a state as it is not storing an estimate for the propositions individually.

If two propositions are known to be mutually exclusive their estimates can be added without compromising admissibility. The h^m heuristic family (Haslum & Geffner, 2000) uses this by calculating the dependance for all m-sized propositions tuples to provide a more informative heuristic. However, any tuple size over 3 is so computationally expensive that it is considered infeasable. An advanced variant of h^m called Boosting (Haslum, 2006) devotes parts of the search to calculate larger tuples for certain propositions, that is the system uses h^2 as its heuristic but for states where the heuristic values are low the system boosts the propositions to h^3 to get a better estimate.

2.3 Summary

The problem of solving a puzzle boils down to one of converting a set of propositions, the initial state, to some other set of propositions, the goal state. The rules are defined using a formal language and consist of the initial state, operators and goal conditions. Operators change one state to the next and are often defined as a tuple, $op = \langle Prec, Add, Del \rangle$. The state transition process in GGP is different from planning in that every proposition has to be propagated to a new state with a *next* relation and thus the Del effects of operators are explicitly defined.

In a *progression space* search, like HSP and FF use, the heuristic must perform the relaxation process every time. This is a time consuming process and quickly becomes the

main bottleneck of such systems. By defining the *regression space*, as was done for HSPr, and storing proposition estimates from a single relaxation, a much faster heuristic can be devised. However, defining the *regression space* without an explicit definition of the negative effects of operators is hard.

One last consideration is that GDL, unlike PDDL, allows negations in the goal conditions. A relaxation process that ignores the negative effects of operators will not work for such puzzles. In a puzzle where the initial state is a large set of propositions and the goal is to have them all negated, ignoring the negative effects of a puzzle will not help at all.

Chapter 3

The Search

In this chapter we describe the single-agent search algorithm we use to solve GGP puzzles. There are several search algorithms that can be used for the problem at hand. Unlike HSP we use a complete algorithm that guarantees an optimal solution given an admissible heuristic and sufficient time. The two most popular such search algorithms are *A** and *Iterative Deepening A** (IDA*). A* works like a breath-first search (BFS) whereas IDA* behaves more like a depth-first search (DFS). The choice between A* and IDA* is one of compromising between memory and CPU overhead. A* requires memory for open- and closed lists but only expands a node once if the heuristic is consistent. IDA* only keeps track of its current path from the start so it must re-expand nodes in its search effort.

The search algorithm we use, Time-Bounded and Injection-based A*, is based on a real-time variant of A*, called Time-Bounded A* (Björnsson et al., 2009). As the name implies TBA* works in a real-time environment which is essential for solving GGP games. The main difference between our TBIA* variant and TBA* is a preference towards re-discovering good paths over back-tracking, and the heuristic injections used to increase the chances of such rediscoveries. This adaptation makes the algorithm better suited for solving GGP puzzles.

3.1 Time-Bounded A*

Real-time search algorithms are bounded by the computing resources they use for each action step by interleaving planning and execution. Most state-of-the-art real-time search methods use pre-computed pattern databases or state-space abstractions that rely on domain dependent knowledge. The Time-Bounded A* (TBA*) is a variant of the A* al-

Algorithm 1 Time-Bounded A*

```
1: solutionFound ← false
2: solutionFoundAndTraced ← false
3: doneTrace ← true
4: loc ← start
5: while loc ≠ goal do
6:    if not solutionFound then
7:       solutionFound ← A*(lists, start, goal, P, N_E)
8:    end if
9:    if not solutionFoundAndTraced then
10:      if doneTrace then
11:         pathNew ← lists.mostPromisingState()
12:      end if
13:      doneTrace ← traceBack(pathNew, loc, N_T)
14:      if doneTrace then
15:         pathFollow ← pathNew
16:         if pathFollow.back() = goal then
17:            solutionFoundAndTraced ← true
18:         end if
19:      end if
20:   end if
21:   if pathFollow.contains(loc) then
22:      loc ← pathFollow.popFront()
23:   else
24:      if loc ≠ start then
25:         loc ← lists.stepBack(loc)
26:      else
27:         loc ← loc_last
28:      end if
29:   end if
30:   loc_last ← loc
31:   move agent to loc
32: end while
```

gorithm that provides real-time response without the need for precalculated databases or heuristic updates (Björnsson et al., 2009). In Algorithm 1 the pseudo-code for TBA* is shown.

Much like A* the TBA* algorithm uses an open- and closed list, where the open list represents the fringe of the search and the closed list contains the nodes that have already been traversed. The algorithm uses A* as its searching method but with a slight alteration. Unlike A* search, TBA* interrupts its planning after a fixed number of node expansions and chooses the most promising candidate from the open list (the node that would get expanded next) and starts to trace the path back toward the initial state. TBA* also keeps two paths, $pathNew$ and $pathFollow$, where $pathFollow$ is the path it is currently following and $pathNew$ is the best path found so far. The agent keeps following

the $pathFollow$ while it traces a new path back. The back tracing halts early if it crosses the agents current location (and updates $pathFollow$), otherwise the path is traced all the way back to the initial state. The agent then starts to backtrack towards the initial state until it crosses $pathNew$ and updates its $pathFollow$. The agent will sooner or later step onto the the path to follow, in the worst case this happens in the initial state.

One special case is if the agent runs out of actions in its $pathFollow$ while waiting for a better path to be traced back. In this case it will just start to backtrack toward the initial state.

3.2 Issues with TBA* and GGP

Although TBA* is effective for many real-time domains, there are a few issues that arise when applying it to GGP puzzles:

- GGP moves are often irreversible,
- GGP only guarantees the goal is reachable from the initial state,
- GGP allows multiple value goals (0-100).

The first issue is that TBA* relies heavily on the agent backtracking as a means of combining the $pathFollow$ and a better $pathNew$. This is not possible in many GGP puzzles as the actions the agent performs are simply irreversible, such as the jump action in the game of Peg. Once the jump is performed a peg is removed from the board and thus the move cannot be undone. Instead of backtracking, we choose to have the agent search his way onto the new and better path. As we will see, this has double benefits as it works for irreversible puzzles as well as finding shortcuts onto the new path in the reversible ones.

Second, GGP only guarantees that the goal is reachable from the initial state, i.e. exploring the state space may lead to unsolvable puzzles. This is obvious in the case of an irreversible puzzles but this also applies to reversible puzzles where a step counter is used to determine the goal value awarded. For example, the 8-puzzle is a reversible game but there is a 60 step limit imposed in the goal condition. Assuming a state where the agent needs 18 steps to reach the goal; the puzzle is still solvable if the state is encountered before time step 43 but unsolvable otherwise. What this means is that in GGP the agent cannot make back and forth moves just to buy time for computations.

Algorithm 2 Time-Bounded and Injection-based A*

```
1:  solved ← false
2:  loc ← start
3:  goalFollow ← 0
4:  while not puzzle.solved(loc) do
5:      Astar(lists, puzzle, timebound, heuristic, loc)
6:      goalNew ← lists.mostPromisingState()
7:      // Is the destination of pathNew better than pathFollows
8:      if goalNew ≥ goalFollow then
9:          // Build pathNew and update pathFollow if possible
10:         pathNew ← traceBack(lists, goalNew)
11:         if crosses(pathNew, pathFollow) then
12:             pathFollow ← updatePath(pathFollow, pathNew)
13:         else if goalNew ≥ injectionThreshold then
14:             // Good but unusable pathNew
15:             heuristic.injectLowHeuristicValues(pathNew)
16:             lists.reset()
17:         end if
18:     end if
19:     // Proximity to Horizon check
20:     if pathFollw.size < 2 or puzzle.irreversible() then
21:         lists.reset()
22:     end if
23:     loc = pathFollow.popFront()
24:     move agent to loc
25: end while
```

Last, due to GGP allowing multiple goal values, the selection of the most promising node has an added criterion of going for the highest goal available if the optimal solution has not been found.

3.3 Time-Bounded and Injection-based A*

We adapted the TBA* algorithm to be better able to handle GGP puzzles. Pseudo-code of the new algorithm, TBIA*, is shown as Algorithm 2. The search is interrupted at a fixed time interval, $timebound$. If the puzzle is not solved in the first time slice the most promising node is selected, see line 6. Due to GGP allowing multiple goal values we use the following three rules in the process of selecting the most promising node:

- an optimal goal has been observed,
- a goal has been observed,

- no goal has been observed.

If an optimal goal has been observed, i.e., one which has a value of 100, the puzzle is solved so the optimal path is stored and followed from then on. If a suboptimal goal has been found we choose to follow that path, but we keep looking for a better one. If, however, no goal has been observed we, like the original TBA*, choose the next node from the open list as the most promising candidate. The logic for this is that to the agent's best knowledge, this is the most promising path available to it.

In subsequent time steps the agent will use $pathFollow$ while still looking for a better path. When a new path, $pathNew$, is discovered there are two questions to answer: is $pathNew$ better than our current $pathFollow$, i.e. does it lead to a higher goal value, and is this new path usable, i.e., does it cross our current $pathFollow$ or are they parallel paths from the starting position (see lines 6-12). When the agent has discovered a new path, one of three conditions holds:

- If the new path is not better it is simply discarded.
- If the new path is better and it shares a common state with our current path, $pathFollow$ and $pathNew$ are merged into a new $pathFollow$, see line 12.
- If the new path is better but the two do not cross, the open and closed lists are reset but $pathFollow$ does not change.

By resetting the lists the search will start with the subsequent state as its new initial state, i.e. state loc moved to at the end of this time step will become the new $start$ (root) for the search in the next time step. Thus the $pathNew$ discovered in the next time step is guaranteed to be usable as it originates from the current location. Note that the state we call $start$ is root of the current search effort and after resetting it is no longer the same state as the initial state of the puzzle.

The agent does not reset the $pathFollow$ and it will continue to follow this path, unless a better usable one is found. This means that the agent is still guaranteed to obtain whatever goal it had already observed.

After the agent has reset its lists it is important to rediscover good paths quickly. We therefore inject low heuristic values for the states of any unusable but better $pathNew$ (see line 15). The value injected will depend on the goal value of the path discovered. This will increase the odds of the agent finding an usable goal scoring path as it will ensure that the search will quickly re-expand the path again as soon as the first low value node is encountered. The agent will rediscover a usable part of the old injected path leading it to the goal with minimum effort. This will work for both irreversible and reversible puzzles

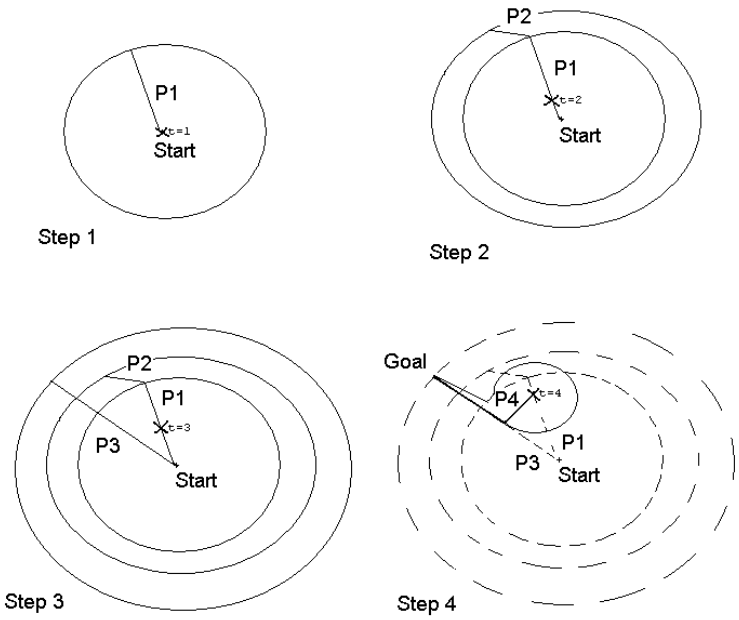

Figure 3.1: TBIA* search space and effects of using resetting

as an alternative path from the current location to the unusable path may exist and for reversible puzzles the worst case is that there is no shortcut possible and the agent must search its way to the $start$ to get on to the better path.

In Figure 3.1, we illustrate how resetting the lists works when a non-usable better path is discovered. The circles represent the search space covered in each time step, the lines are the paths and the X is the agent's current location. In time step 1 the path P1 is the most promising and becomes $pathFollow$. In time step 2 the path P2 is the most promising and happens to extend the current $pathFollow$, so they are joined to a new and extended $pathFollow$. In time step 3 an unusable better path P3 is discovered at which point TBIA* resets the search effort and injects low heuristics values for the states on P3 but keeps moving along $pathFollow$. In time step 4 we see the old search space as dotted circles and the new search space as solid. The discovered usable path P4 is a shortcut onto the previously discovered unusable path P3. The spike in the search space is the result of the search running down the injected path toward the goal with minimal effort. The discovered path P4 ($pathNew$) is guaranteed to be usable as the current location of the agent is the root of the search. The $pathFollow$ is updated, or essentially replaced with $pathNew$, and the puzzle has been solved. If, however, the goal is not an optimal one, the agent will continue to try to discover a better goal with continued search.

In reversible puzzles with deep solutions the agent can run into trouble as it approaches the fringe of the search. With every time step the fringe of the search expands less and

less with regard to solution depth. If the initial path is almost used up and the agent has not discovered a better usable path it may be better to reset the lists and focus the search on the surrounding area as it will have to fall back on making random actions if the path ends. Another validation for applying resetting more frequently is that in GGP competition puzzles the allowed steps are often limited so focusing the search after a few steps increases the odds of finding a better solution as it may not be feasible to perform many backtracking moves to get onto the solution path.

If the puzzle at hand is irreversible the lists of TBIA* should be reset frequently (even as often as after each step) as there are no guarantees that the agent can make the crossing to any better parallel path. Much of the search effort is thus spent on expanding potentially unreachable states. In our experimentations and in the 2008 GGP competition the agent reset the lists after each step if the puzzle was discovered to be irreversible.

3.4 Summary

The TBA* algorithm forms the base of our new real-time algorithm, TBIA*. The algorithm handles multi-valued goals and irreversible moves, needed for solving GGP puzzles. We have the agent re-search his way onto previously unusable, but good, paths instead of back-tracking. To increase the odds of rediscoveries a low heuristic value is injected for the states on such paths.

At the heart of any informed search method there must by an informative heuristic. How we derive such a heuristic for GGP puzzles is the topic of the next chapter.

Chapter 4

Deriving a Heuristic

The purpose of a heuristic function is to provide an estimate of how far a given state s is from a state $goal$, where the puzzle is solved. To derive this estimate the heuristic function solves a relaxed problem and uses the length of the relaxed solution as a lower-bound estimate as to the true length of a solution to the original problem. This is all good and well if one knows how to make a 'good' relaxation of the problem. A good relaxation must be easier to solve than the original problem, but still remain informative. If the lower-bound estimates are too low, or uninformative, the search makes slow progress as it can not prune off undesirable branches in the search tree. This is commonly referred to as thrashing.

4.1 The Relaxation Process in Theory

The relaxation process that we use for GGP is similar to HSP's (Haslum & Geffner, 2000), and is based on ignoring:

- the restriction of applying one action per time step,
- the negative effects of actions.

By ignoring the restriction of choosing a single action at each time step the relaxation process is rewarded with a much wider perspective in that it quickly accumulates facts, but this is at the cost of knowing the exact solution path as there is no longer any way to know which of the actions were necessary for satisfying the goal condition. This is acceptable as the relaxation process is not looking for the solution path, but rather estimating how far the current state is from a goal. This modification alone will not simplify the problem as the additive effects of one action may be countered by negative effects of another. The

Algorithm 3 Heuristic function

Require: The goal set G, current state s, integer $Steps$ and integer $UnsatisfiedGoals$

1: *// Penalty calculated*
2: $UnsatisfiedGoals \leftarrow 0$
3: **for all** $p \in G$ **do**
4: **if** $p \notin s$ **then**
5: $UnsatisfiedGoals \leftarrow UnsatisfiedGoals + 1$
6: **end if**
7: **end for**
8:
9: *// Distance calculated*
10: $Steps \leftarrow 0$
11: $s_{super} \leftarrow s$
12: **while** $G \not\subseteq s_{super}$ **do**
13: $A_{curr} \leftarrow A(s_{super})$
14: **for all** $a \in A_{curr}$ **do**
15: $s_{super} \leftarrow s_{super} \cup Add(a)$
16: **end for**
17: $Steps \leftarrow Steps + 1$
18: **end while**
19: **return** $UnsatisfiedGoals + Steps$

second modification needed is to ignore all the negative effects of actions, i.e. any undoing or removal of propositions, and let the propositions accumulate into a super state until a goal is satisfied. This relaxation process is illustrated in Algorithm 3. How the wider perspective works can bee seen in line 14 of the algorithm, where the super state is grown by adding to the state s_{super} all positive effects $Add(a)$ of all the actions available $a \in A_{curr}$ in the current time step, until the super state contains all the necessary propositions to satisfy a goal condition. The derived time step counter, named $Steps$, is how many steps are necessary to reach a goal. As we ignore the negative effects of actions we lose any negative interaction that might occur when multiple actions are performed in the same step. The derived estimate is thus a lower-bound on the true length of the necessary sequence of actions to take the agent from the estimated state to a valid goal state.

In the heuristic we use a maximization over all the proposition estimates, g_s^{max}, as the $Steps$ measures the relaxation steps required for the hardest proposition. Remember that HSP assumes full independence between propositions and uses g_s^+ and adds all the propositions to calculate the state estimate. We do not assume full independence between proposition but we do want to take into account how well the agent is doing with regard to how many goal propositions it has already achieved. A penalty of 1 is added for every goal proposition not satisfied resulting in possibly overestimating the true distance. However, this overestimate is upper-bounded by how many goal propositions, exceeding one,

can be achieved by a single action. If all propositions require at least a one action to be achieved there can be no overestimate. As our agent has limited time when competing in GGP and it is important to find goal scoring paths quickly, we are willing to sacrifice admissibility for a more informative heuristic.

The goal conditions in GGP are usually represented as formulae in the game descriptions. To be able to apply the additional penalty for unsatisfied goal propositions the set of goal propositions G needs to be derived. This is done the first time the relaxation process finds a goal[1] in the relaxed super state s_{super}. To find the goal propositions a brute-force algorithm is used: for each $p \in s_{super}$ remove p and check for goal. If goal still holds move on, else add p to the state again and move on. At the end of this process only the required propositions to satisfy the goal remain. The goal set G is stored and used to calculate the penalty for consequent state estimations.

We are certainly oversimplifying here as GGP allows for multiple goal conditions and this is only one of many possible goals. But it is the closest goal according to the relaxation, and the one the relaxation process estimate is based upon. Another way to look at this is that this is the best we can hope for. Remember that the full search is not using relaxation so if the heuristic is wrong it just means that the heuristic is uninformative or in the worst case misleading. The estimates are most probably lower-bound approximations to the true distance. Underestimating will cause thrashing and hence slower search but not affect solution quality.

The relaxation process is time consuming as there is much work involved to generate the states, state transitions and all the available moves at every expansion step. A way to deal with this is to store individual propositions and their value estimate in memory. As was explained in the discussion about HSPr, if a proposition estimate can be derived over how far it is from the goal it only needs to be computed once. This topic will be discussed in more detail later in this chapter.

4.2 The Relaxation Process in Practice

Unfortunately, like so many other things, the relaxation process does not work as well in practice as it does in theory. As an example of the practical difficulties in applying the relaxation process we use the 8-puzzle. In Figure 4.1 we show the first few relaxation steps of the puzzle. The double-lined grid separates the 9 squares of the puzzle and the

[1] Note that only the maximum score of 100 is considered a goal here even though GDL allows for multiple goal values (0-100).

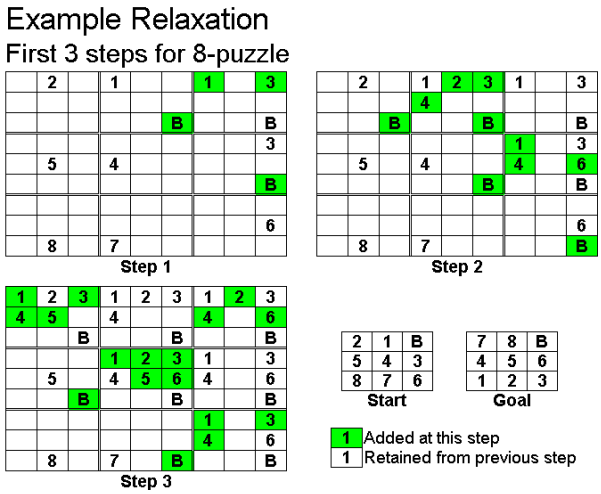

Figure 4.1: First three steps of relaxing 8-puzzle according to the theory

single-lined grid indicates which tiles are present in each square. In the top left corner we see what the super state looks like at step one, with non-colored squares being the initial state and the colored squares are the propositions added by the available actions. For example, in step 1 the blank square (B) can be moved either left or down switching places with 1 and 3, respectively. One way to look at what happens in the relaxation process is that the tiles of the puzzle get stacked in each square, so more numbers indicate a bigger stack. The initial state given in the example results in a seven step relaxation process at the end of which every tile is in every square. This is because the negative effects, i.e. removing propositions from the current state, are not performed. The tiles with a colored background indicate the propositions introduced to each square in the current relaxation step. The goal is not reached until the tile 7 is propagated to the top left corner and as said before this happens in the seventh relaxation step. The heuristic estimate according to g_s^{max} is thus 7 for that is the value assigned to the atom $(cell\ 1\ 2\ 7)$ as well as the step count for the relaxation process.

In practice the relaxation of 8-puzzle only requires five steps. The author of the puzzle's GDL description correctly assumes that there can only be one blank tile in any given state and as a result the *next* relation that propagates moved tiles to the next state will do so for any square containing the blank tile in the same row or column. To clarify this we can see in Figure 4.2 the second step of the relaxation process in practice. Notice how the tiles 2 and 6 have been propagated to the top right corner square because it still contains a blank tile from the initial state and we never remove any tiles. This causes the tiles to spread too quickly and the process has reached the goal condition in only five steps instead of seven. If the *next* relation were altered to only propagate tiles to the neighboring squares

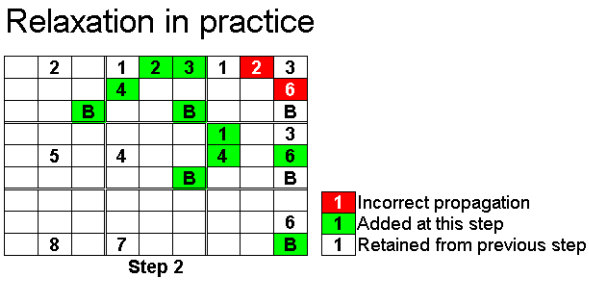

Figure 4.2: The second step of Relaxation in practice

this would not happen. The author's assumption of only one blank tile does not hold in our relaxed rules and as a result we have a less informative heuristic.

How the author of a puzzle chooses to formulate its description will significantly impact the informativeness of relaxation or even make it impossible. A simple example would be to make a non-terminal state a precondition for all actions. If the relaxation's super state satisfies a terminal condition before a goal condition there will be no action to further progress the super state and the relaxation process must give up.

Negations violate the second modification of ignoring delete effects. Any negated precondition for a necessary action or in goal conditions will result in relaxation failure. In puzzles where the goal condition can not be satisfied in a relaxation of the initial state or we choose not to derive the goal set (we only allow the derivation of a goal set when the super state is sufficiently small or it would take too much time), we may still want to provide some informed estimate. We use the following rule:

$$E(s) = \begin{cases} Count & \text{if } goalValue \text{ is } 100 \\ 100 - goalValue(s) - Count & \text{otherwise} \end{cases}$$

where $E(s)$ is the derived estimate for state s. The following rule is derived from the assumption that we want to keep looking for a full score goal as long as possible and if we find one we want go get to it in the least steps possible.

Then there are puzzles where the ideas of relaxation just do not capture the complexity of the puzzle, such as the Lightson puzzle (see Section 5.4 in Chapter 5), where the relaxation always derives a time step counter of 1 as every action in the puzzle is available from the initial state.

In many puzzles the GDL representation includes a step counter within the puzzle. A step counter poses a problem for the heuristic as it is often part of the goal condition, or even the only goal condition. In cases where the goal value is determined, at least in part, by

the state of a step counter the relaxation process may satisfy other goal conditions before the step counter, i.e., solve the problem in less then the required solutions steps. This should not be possible, but as the relaxation process has changed the rules by performing multiple actions at every time step, it can.

4.3 Proposition Caching

As with the original HSP planner the relaxation process of the heuristic is the main bottleneck of the search effort. This due to *progression space* search having to perform relaxation of every state. HSP's solution was a different variant they called HSPr were the relaxation estimates where derived only once in the *regression space*, starting from the goal conditions and then as propositions are added their distance is stored in a table for future lookup. This process halts when all the propositions required for the initial state have been estimated. The search is then performed in the *progression space* as before but now the heuristic is derived quickly from the stored proposition estimates in the lookup table.

In GGP it is not possible to define the *regression space* and thus a single regression relaxations is not possible. This is due to how the state transition is performed in GGP, where the *next* predicate derives the next state and this is hard to reverse. However, we can approximate the derived lookup table by storing distance values for individual propositions as they are discovered.

4.3.1 The Caching Mechanism

The aim of the caching mechanism is to store, in a fast lookup table, the distance of all propositions to a goal. Whenever we perform a successful relaxation of some state s the relaxation step counter $Step$ indicates how far the hardest of its propositions is from a goal, $Step = max(p \in s)$. Unfortunately we do not know which of the state's propositions this applies to. We can assume that they were all the hardest proposition, until proven otherwise, and store each proposition with the relaxation step count value. Then for every successful relaxation that is performed we use the following update rule:

$$Initialize\ E(p)\ to\ \infty \tag{4.1}$$

$$For\ each\ p\ \in\ s\ perform:\ E(p) = Min(Step, E(p)). \tag{4.2}$$

Algorithm 4 Heuristic function with Cache

Require: The goal set G, current state s, integer $Steps$, integer $UnsatisfiedGoals$ and cache set E

```
1: Steps ← 0
2: UnsatisfiedGoals ← 0
3: CacheMiss ← False
4: // Penalty calculated
5: ... Same as before
6: // Check if all propositions have cached values
7: for all p ∈ s do
8:     if p ∈ E then
9:         Steps ← Max[Steps, E(p)]
10:    else
11:        Steps ← 0
12:        CacheMiss ← True
13:        Breake loop
14:    end if
15: end for
16: if CacheMiss then
17:    // Distance calculated
18:    ... Same as before
19:    // Update the cache
20:    for all p ∈ s do
21:        if Steps < E(p) then
22:            E(p) ← Steps
23:        end if
24:    end for
25: end if
26: return UnsatisfiedGoals + Steps
```

where $E(p)$ is the stored proposition estimate and $Step$ is the step count value derived from the relaxation process.

If any proposition in the state to be estimated does not have a cache value a relaxation process is invoked. Note that we only update the values for the original state propositions and not the propositions generated in the relaxation super nodes. Storing values for the propositions of the super node would almost certainly reduce the number of relaxations required but after the first expansion, due to the relaxation method, most of the added propositions would actually not be leading us towards the goal (as we add all the available actions but only few of them are helping). This could lead to a gross underestimate of non-helping propositions. For example, all information about dead-end states could be lost.

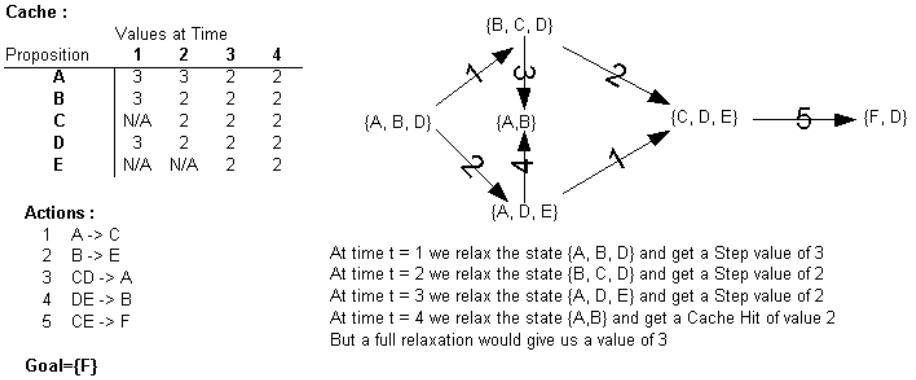

Figure 4.3: Proposition dependency is lost with the cache hit of state {A, B}

The stored proposition estimates will converge to their true distance if enough relaxations are performed, but how many is enough? This depends on how propositions are added and removed as the puzzle progresses and hence is very domain dependant. Too few relaxations can result in overestimation of states and can degrade the solution quality. Low estimates result in more thrashing and hence slower search progress. The speed-up of state cache hits versus relaxing them is very significant and some puzzle are only viable to solve when cache is used.

Although convergence cannot be guaranteed we can increase the probability by making relaxations more frequent. For example, by introducing a threshold k such that after some fixed amount of cache hits a relaxation is always performed. This enforcement is thus a compromise between speed and heuristic quality.

One of the drawbacks of the proposed caching mechanism is that it suffers from lost information about proposition dependencies as the propositions only provide information about the state where they had the lowest possible value. This is best illustrated with an example, see Figure 4.3. Assuming a puzzle where propositions A and B are in the initial state and each is a necessary precondition to satisfy actions leading to the goal. A relaxation of a state {A, B, D} results in an time step count of 3 and hence A, B and D are stored in the cache with estimates of 3. By applying action 1, A achieves C, the next state {B, C, D} will relax with a step count of 2 and hence the estimates B and D are updated to a value of 2 at time 2. If we then apply action 2, B achieves D, and will relax the state {A, C, D} we again get a step count of 2 and update the estimate of A to 2. At this point, after time 3, both the proposition estimate for A and B have been updated to a value of 2 and now when the state {A, B} is estimated it will get a cache hit with with a value of 2 as $Max[E(A), E(B)] = 2$. Now if we were to relax the state, the true estimate should

be 3, just as it was for the state {A, B, D}. The independent estimates for propositions A and B have converged to the correct value 2 but the information that if both A and B are part of a state their joint estimate should be 3 is lost. Underestimating does not affect the solution quality but it will result in the search thrashing, as it needs to expand more nodes to find a solution.

4.3.2 Adaptive Caching

As discussed above, the caching implementation uses a variable k that indicates how many cache-hits it will allow before forcing a relaxation. This count is reset to zero every time a relaxation is performed, forced or not. k is used to increase convergence of estimates and prevent overestimates. Instead of using a fixed k, one can adapt it during the solution process. We propose such a variant and choose to call it $adaptive\ k$. The $adaptive\ k$ performs an additional check at the end of relaxations where it compares the relaxation value derived and the cached value if it was available. Depending on the result the threshold k is increased or decreased making relaxation more or less frequent. This is an attempt to get the best of both caching and always relaxing, controlled by the observed quality of the cached values. How much to increase and decrease the threshold value remains an open problem. In the results presented here the threshold starts with a value of $k = 4$ and is incremented by 4 for ever time the values match. If they do not match, however, k is decremented by 25%. This makes the mechanism fall back on relaxing frequently for domains where caching is not providing accurate estimates.

4.4 Summary

The heuristic estimates how far a given state is from a goal state by combining two estimates. The first is the distance estimate derived from solving a relaxed puzzle and the second is a penalty that is added for each goal proposition which is not satisfied. As a single action can achieve more than one proposition the resulting estimate is non-admissible, but the combined value is more informative than only using the first part.

How the description of the puzzle is stated has significant impact on the relaxation process. Many puzzles have little or no information embedded in their goal conditions and if they also are unsuitable for relaxation the heuristic can be uninformative. However, this process provides more information about a problems complexity than no process at all and for many puzzles, where the relaxation process is well suited, it performs quite well.

Even when the relaxation process does not find a goal it can provide some information about how many relaxations steps are performed before a terminal state is reached. The problem is then to interpret this result, i.e., should the agent avoid such paths, assuming they are dead-ends, or pursue them in the hope that they are terminal goal states that the relaxation cannot interpret correctly.

Chapter 5

Empirical Results

In this chapter we present our results from solving several puzzles. We will see examples of different behaviors and difficulties the agent must face.

5.1 Setup

The computer used to obtain the results is a MacBook with a Intel Core 2 Duo 2.2Ghz CPU with 4MB L2 cache and 2.5GB DDR2 667Mhz RAM. The results are from two variations of testing, an unbounded time variant and a real-time variant using competition like settings. With the unbounded time variant we establish how much time, in seconds, is required to solve the puzzle from the initial position. As this is often time consuming, the results are only based a single solution. In a competition the agent has a start and a play clock and after each play clock an action must be performed. As a baseline for all puzzles we use 10 seconds for both clocks. For the harder puzzles we also use longer start clocks as was done in the GGP competitions of 2006 and 2007. The time allocated for the start clock can thus vary from 10 seconds to 180 seconds. We should note that the first action of the agent is not performed until the start clock plus the first play clock are up. We will call this the time before first action in our discussion of the different puzzles. All real-time results are the average of running each algorithm 10 times.

We are testing TBIA* with four variants of the caching threshold k:

- $k = 0$, relaxation is always performed
- $k = 40$, a maximum of 40 cache-hits allowed in a row before relaxing
- $k = 200$, a maximum of 200 cache-hits allowed in a row before relaxing

Table 5.1: 8-puzzle results with no time limit

Optimal solution length: 14							
Threshold	Seconds	Closed l.	Open l.	Cache hits	Relax.	Moves	Score
$k = 0$	10.0	127	101	0	231	14	100
$k = 40$	2.5	173	123	252	52	14	100
$k = 200$	2.5	173	123	253	51	14	100
Adaptive k	2.8	173	123	245	59	14	100
UCT	10200.0	n/a	n/a	n/a	n/a	60	0
Optimal solution length: 18							
Threshold	Seconds	Closed l.	Open l.	Cache hits	Relax.	Moves	Score
$k = 0$	206.7	2029	1407	0	3567	18	100
$k = 40$	13.4	2111	1316	3421	149	18	100
$k = 200$	9.8	2096	1310	3461	85	18	100
Adaptive k	12.9	2099	1320	3419	140	18	100
Optimal solution length: 30							
Threshold	Seconds	Closed l.	Open l.	Cache hits	Relax.	Moves	Score
$k = 0$	8074.0	85915	29878	0	136322	30	100
$k = 40$	464.2	104841	26590	154434	4024	30	100
$k = 200$	209.8	77691	24775	119204	67	30	100
Adaptive k	2240.5	76580	25614	83973	35170	30	100

- $Adaptive\ k$, where the quality of cache estimates controls the threshold.

We also compare the performance to the simulation based algorithm used in CADIA-Player (Finnsson, 2007). The simulation algorithm uses a Monte Carlo rollout with upper confidence for trees (UCT) (Kocsis & Szepesvári, 2006) and is hence referred to as UCT. To keep things fair the UCT results are obtained with the same processing power as TBIA* on a single CPU. In competition CADIA-Player can use multiple machines for its random rollouts. It should be noted that UCT is first and foremost designed for multi-role advisory games although some enhancements have been made to deal with single agent games as well.

5.2 8-puzzle

The 8-puzzle is a 3x3 sliding-tile puzzle where the objective is to have all the tiles in some particular order. The puzzle is reversible and there are three goals defined in the GDL. A score of 100 is awarded for an optimal solution, and a score of 99 is awarded for a solution in less than 60 actions. Finally, a score of 0 is given for 60 actions without a solution. The complexity of the puzzle can be set by different starting positions.

Table 5.2: 8-puzzle results with time bounds

Optimal solution length: 30						
Threshold	S.C.	P.C.	Cache hits	Relax.	Moves	Score
$k = 0$	10	10	0	11707.2	59.2	29.7
$k = 40$	10	10	150329.6	4830.7	45.2	89.1
$k = 200$	10	10	240197.2	2126.8	43.2	99.0
Adaptive k	10	10	75480.1	7461.7	48.2	89.1
UCT	10	10	n/a	n/a	60.0	0.0

In Table 5.1 we see results from using three starting positions with optimal solution length 14, 18 and 30 actions. As the UCT algorithm could not solve the simplest of the puzzles it was not run on the two harder instances.

Note that with the start and play clocks set to 10 seconds the time before first action is 20 seconds and as we see in Table 5.1 all the caching variants of TBIA* except $k = 0$ require less time to solve the two simpler instances. What this means is that there will be no real-time search and thus no need to test these variants in competition mode for those instances. Only the $k = 0$ variant cannot solve the instance with solution length 18 within the time before first action and when run in so it was run in the competition settings it could not solve the puzzle.

The caching mechanism works well in the 8-puzzle and the more cache hits we allow the quicker a solution is found. The speed increase of using caching more than makes up for any thrashing it may cause. The representation of the problem works very well for how the caching works. A state in the 8-puzzle is represented as a list of the positions of all 8 tiles on the board and the goal is to have all tiles in some defined goal position, in our case in alphabetical order. Given that a tile, lets say A, is currently in the bottom right corner, $cell(1\ 3\ A)$, the agent needs to perform a minimum of 4 moves to get it to its correct place in the top left corner, $cell(3\ 1\ A)$. As this is the largest Manhattan distance possible on a 3x3 board this value is derived with good accuracy and no change in any other state proposition will ever make it otherwise. The remaining propositions in this state are also assigned an estimate of 4. That results in possible overestimates until the values converge to their correct Manhattan distance.

A detailed study of cached vs. relaxation estimates for all 303 calls to the heuristic in the 8-puzzle of solution length 14 shows that in 69.3% cases the two values are the same. In 14.9% cases the cache over estimates and in 15.8% cases it underestimates. The highest cache over-estimate was only 2 higher, and the lowest under-estimate was 3 lower then the derived relaxation value. The sum of all 303 estimates for relaxation is 3332 and the

Table 5.3: Analysis of Cache vs. no Cache

Evaluations	No Cache = Cache	No Cache < Cache	No Cache > Cache
Match ratio	69.31%	14.85%	15.84%
Maximum difference	0	2	3

cache hit sum is only 7 higher or 3325. The cache vs. no cache results are summed up in Table 5.3.

5.3 Peg

Peg is an irreversible game where pegs are removed as another peg jumps over them. In the beginning there are 32 pegs on the board, arranged in an cross-like arrangement with the center hole being the only one empty. The game terminates when there are no moves available and the goal value is calculated depending on how many pegs remain on the board. A score of 100 is awarded for 1 peg in the center hole and 99 is awarded for 1 peg in any other position. For each peg exceeding 1 the score is decremented by 10, i.e., for 2 pegs remaining the score is 90, for 3 it is 80, etc.

Due to the nature of the game, with high branching factor and a depth of 32 moves, it has a large state space. Solving Peg on the start clock takes a long time, no matter what the k threshold is set to. Instead we only use competition settings. Note that due to Peg being an irreversible game, once an action has been selected and performed, all the other actions and subsequent states are discarded from the open list as there is no guarantee that the agent can ever get to them. What this means in practice is that the TBIA* resets its open- and closed list, as well as the proposition cache, after every action performed. One of the advantages is that the search gets focused on its current neighborhood and incorrect cache values obtained from some other branch of the game tree are discarded.

In this puzzle the caching variants have the problem of the independent proposition estimates converging to low values, causing the combined information to be lost. This leads to dead-ends looking promising and the search is thrashing. It is also informative to note that this issue with the convergence can get worse when the the start clock is longer as the heuristics converges to lower values before the agent is forced to commit to the first action. The problem in Peg is that low scoring branches will get estimated as feasible and the search will spend time looking in low scoring parts of the game tree.

Table 5.4: Peg results with time bounds

Threshold	S.C.	P.C.	Cache hits	Relax.	Moves	Score
$k = 0$	10	10	0	847.1	28.5	75.0
$k = 40$	10	10	3590.2	621.1	28.9	79.0
$k = 200$	10	10	4432.4	485.6	29.0	80.0
Adaptive k	10	10	736.0	752.2	28.4	76.0
UCT	10	10	n/a	n/a	30.0	90.0
$k = 0$	180	10	0	1391.8	30.0	90.0
$k = 40$	180	10	16208.5	1508.9	29.9	89.0
$k = 200$	180	10	25716.5	635.8	28.2	71.7
Adaptive k	180	10	5035.5	1576.0	30.0	90.0
UCT	180	10	n/a	n/a	30.0	90.0

5.4 Lightson

Lightson is a game where the objective is to press buttons to turn on a sequence of lights. The original problem has 4 lights in a 2x2 grid, and defined as $cell(x\ y\ [0..4])$ where 0 is off and 1-4 means light is on. For example, if the button for cell 1 1 is pressed a proposition $cell(1\ 1\ 4)$ is added to the state. The last value, 4, will then decrease with every action made, so the correct buttons must be pressed in sequence although the order does not matter. The Lightson2x2 is a set of 4 original problems in a 2x2 grid, giving a total of 4x4 buttons. To solve the Lightson2x2 puzzle only one of the original problems needs to be solved. The Lightson4x4 is a set of 16 original problems or a grid of 8x8 buttons. In this puzzle a full score of 100 is awarded for any solution in 10 actions or less.

The Lightson puzzle has some interesting properties. First there is the problem of the increased branching factor caused by having to solve any of the sub-puzzles. As we do not discover the problem's independence of each other, the search must expand all the available actions. If we recall how the relaxation works, by allowing the relaxation phase to apply all available actions, the step count derived from relaxation will be uninformative as it is always 1. Not only does the relaxation solve one of the combined lightson puzzles, but it solved all of them and with every light indicated as on by the value of 4. What this means is that the cache value for all propositions will converge to a value estimate of 1, as there was only one relaxation step and all actions were available, and thus the first part of our heuristic is not informative. The second part of the heuristic is also affected as we reward only cell propositions where the last parameter has a value of 4, that is only the most recently light turned on. When the agent presses the second button the first proposition will have degraded to a value of 3.

Table 5.5: Lightson results without time limits

Lightson2x2, optimal solution length: 4							
Threshold	Time	Closed l.	Open l.	Cache hits	Relax.	Moves	Score
$k = 0$	9.2	89	1320	0	1409	4	100
$k = 40$	29.6	1325	19428	20615	569	4	100
$k = 200$	27.3	1325	19428	21026	158	4	100
Adaptive k	3.1	137	2040	2088	89	4	100
Lightson4x4, optimal solution length: 4							
Threshold	Time	Closed l.	Open l.	Cache hits	Relax.	Score	
$k = 0$	937.4	328	20595	0	20929	4	100
$k = 40$	10680.0	20314	1276036	1267375	32650	4	100
$k = 200$	9713.0	20314	1276036	1293319	6706	4	100
Adaptive k	9425.0	20314	1276036	1298976	1049	4	100

Table 5.6: Lightson4x4 results with time bounds

Threshold	S.C.	P.C.	Cache hits	Relax.	Move	Score
$k = 0$	10	10	0.0	2552.6	10.0	0.0
$k = 40$	10	10	5759.6	608.0	7.0	100.0
$k = 200$	10	10	5338.9	342.7	5.7	100.0
Adaptive k	10	10	6341.2	450.8	6.4	100.0
UCT	10	10	n/a	n/a	8.6	60.0
$k = 0$	30	10	0	2538.9	8.4	70.0
$k = 40$	30	10	8260.6	608.0	7.0	100.0
$k = 200$	30	10	9340.3	402.8	5.0	100.0
Adaptive k	30	10	8124.4	357.6	5.6	100.0
UCT	30	10	n/a	n/a	9.1	30.0

In Table 5.6 are results from competition settings. There are two time settings used, the default 10 second start and play clock and a 30 second start clock and 10 second play clock.

Given the competition times of 10 second start and play clocks the $k = 0$ version suffers due to how few nodes it expands. The first action, after 20 seconds, is based on only 9 node expansions. Subsequently the $k = 0$ variant was not able to solve the puzzle in any of the 10 runs. The $k = 40$, $k = 200$ as well as the $adaptive\ k$ variants manage to get full scores, but not by finding the optimal solution length of 4 actions. If the start clock is increased to 30 seconds, play clock still at 10, the $k = 0$ can solve the puzzle most of the time and the $k = 200$ is getting close to the optimal solution length.

We cannot state that any of the variants is better than the others as the main issue of this puzzle lies with the quality of the heuristic. The heuristic derived by relaxation is poor in this puzzle, and it shows clearly how the GDL description can have a big effect on how

informative the heuristic will be. We are neither able to capture a step count estimate, as we do all the actions on a single step, nor the goal propositions due to the relaxed state containing only cell values with the last variable of value 4, a state that can not be reached in the search. Nonetheless we are doing better than the UCT simulation-based counterpart due to our systematic search effort and the short solution depth of only 4 steps.

5.5 Coins2

Coins2 is a puzzle about stacking a row of coins. The agent starts with 20 coins and is allowed to stack coin x on to coin y if both x and y are not already part of a stack and there are two coins between them. A score of 0 is awarded if some coin remains unstacked, and a score of 100 if the agent performs 10 actions, and hence has stacked all the coins. However, due to a flaw in the game description a goal can be reached in only 9 actions, leaving 2 coins unstacked.

The results are shown in Tables 5.7 and 5.8. The advantages of caching is quite large. As expected, the caching variants need less time, but note that they also explore fewer states, i.e., $k = 200$ is 'too good'. By adding relaxations and cache-hits for the $k = 200$ variant we see that only 3867 states were explored while the $k = 0$ variant explored 1198335 states. As they are using the same search method and not performing real-time actions the difference must be the derived heuristic values. Now how can it be that $k = 200$ is deriving a more informative heuristic? What happens is that it takes some time for the proposition's estimates to converge to their correct value. The less frequent relaxations result in slower convergence and until the values converge the overestimated states are working as a pruning method favoring recently observed states, i.e. states where new propositions have forced a relaxation and propositions have moved closer to their correct estimates. As there are several ways to solve this puzzle the branch that just happens to get more focus will eventually lead to a solution. Note that the order of the stacking is not relevant to the solution, so many of the propositions will converge to an estimate of 1 much like what we observed in the Lightson puzzle. Once the cache estimate values converge, the caching is actually worse off then the $k = 0$ mechanism as it will have cache hits for many of the dead-end paths the $k = 0$ will pick up on. As the cache estimates converge to low values and the cache misses dead-ends, the relaxations and cache hits will rarely agree and the $adaptive\ k$ will behave much like the $k = 0$.

Note that the 180 seconds is more than the $k = 200$ requires to solve the puzzle before the first action is taken. It is also interesting to observe that the $k = 40$ mechanism gets a full

Table 5.7: Coins2 results without time limit

Optimal solution length: 9							
Threshold	Seconds	Closed l.	Open l.	Cache hits	Relax.	Moves	Score
$k = 0$	10201.0	515608	210115	0	1198335	9	100
$k = 40$	2646.0	181685	401843	774383	84508	9	100
$k = 200$	41.7	444	2756	3717	150	9	100
Adaptive k	7511.0	480769	541618	924215	657328	9	100
UCT	2045.7	n/a	n/a	n/a	n/a	9	100

Table 5.8: Coins2 results with time bounds

Threshold	S.C.	P.C.	Cache hits	Relax.	Moves	Score
$k = 0$	10	10	0	2703	7	0
$k = 40$	10	10	5471	529	9	100
$k = 200$	10	10	2807	239	9	100
Adaptive k	10	10	1822	1291	7	0
UCT	10	10	n/a	n/a	9	100
$k = 0$	180	10	0	22251	9	100
$k = 40$	180	10	52706	6860	8	0
$k = 200$	180	10	3717	150	9	100
Adaptive k	180	10	17994	17014	9	100
UCT	180	10	n/a	n/a	9	100

score with 10 second start clock but fails to solve the puzzle given the 180 second start clock. This shows clearly how the behavior of the caching mechanism changes once the proposition estimates have converged and it starts to miss the dead-ends. It can be seen in the results for $adaptive\ k$ that the threshold remains quite low as just over half of the calls to the heuristic are cache hits.

5.6 Other Puzzles

Incredible is a blocks world puzzle where the goal is to build two towers and leave the gold in a certain place. The cache has the same problems as with Coins2 where it is prone to use cache hits on dead-end paths. The $k = 200$ however makes up for the extra work by much faster search, and solves the puzzle in the shortest time, 225.7 seconds. In competition settings all the mechanisms get a full score.

Lightsout is a puzzle that was used in the 2008 preliminary GGP competitions. None of the competitors was able to solve the puzzle. The puzzle has a solution depth of 10 actions and is similar to Lightson, just this time the goal is to turn off all the lights. This goal conditions is enforced by negating a light on check, and hence cannot be relaxed. To

be able to use this puzzle the negation was removed from the goal condition and replaced by an exhaustive list of propositions, *(cell1 1 off), (cell 1 2 off)* etc. In all fairness it should be pointed out that checking this list of conditions is much easier for our prover compared to instantiating a *(cell x y on)* and hence the whole search process benefits from thins change. After this change the puzzle is easy to solve, 83.2 seconds by $k = 0$ and only 14.2 seconds by $k = 200$. When run in competition settings with 10 second start and play clocks the $k = 0$ did not solve the puzzle but all the other three already solved it before the first action was taken.

Eulersquares is a complicated puzzle where the objective is to color a partitioned area with three colors without two neighboring subsections having the same color. The game terminates after 30 actions and the score is determined by how many subsection pairs have been colored, ranging from 100 for all 9 pairs to 11 for 2 pairs. In competition settings, 10 second start and play clocks, all the variants solved the puzzle but $adaptive\ k$ solved it in 25 actions while the others solved it in 30.

Chinesecheckers is a single player version of the Chinese checkers game. It is not a legal GDL as the goal can diminish during play. In the current rules for GGP, once a agent has observed a state with some goal it can not get less from subsequent states. We include it here because it is relaxable and can still give information as to what mechanism works best. The winner in this puzzle is $k = 40$ with a solution in 27.2 seconds, and all three caching mechanisms solving the puzzle in less then half the time it took $k = 0$. In competitive settings, 10 second start- and play clock, the $adaptive\ k$ was the only one to find a solution in 11 steps with $k = 0$ and $k = 200$ on step behind or in 12 steps.

Asteroids is a puzzle where the agent must navigate a spacecraft to a planet. There are two modifications to this puzzle, two puzzles in a row called Asteroids_serial and a two puzzles solved in parallel called Asteroids_parallel. Overall the $k = 200$ mechanism works best in the Asteroids puzzles even thought the caching mechanisms are running into some trouble.

In addition to these we have tested nine other puzzles where the relaxation process works. In all nine the caching variants solve the puzzles within within 2 seconds and are thus too simple to provide a productive evaluation of the effectiveness of our methods.

5.7 Summary

The search speed gained with the caching mechanisms is obviously helping in many of the games tested and some of them: 8-puzzle, Lightsout (non-negated), Asteroids and

asteroids_serial, are only solvable with caching. Performing relaxations every time is just too slow and there is often little knowledge behind the first action taken. The risk is that the agent will need to backtrack to undo the first move or in the worst case it may have thrown away its chance of obtaining a full score.

The cache however has certain problems, such as converging to low value estimates quickly, missing dead-end paths due to cache hits and overestimating states if convergence is too slow. As we saw in the Coins2 puzzle there are cases where slow convergence helps, as it focuses the search effort. However, this kind of effect can not be counted on as it might as well be focusing the search on a poor part of the game tree.

The $adaptive\ k$ mechanism is a promising idea that warrants further research. It compromises between relaxations and caching based on the quality of the cache hits. $adaptive\ k$ can however not handle puzzles where the cache and relaxation agree frequently, but the few times they do not agree are the crucial states leading to solutions. Cache hits will have become frequent and the odds on relaxing the crucial states thus low.

In real-time the agent must commit to some branch of the search tree after the time before the first action, i.e. when it performs its first move. Committing to a branch of the search tree narrows the search effort, especially if the puzzle is irreversible. Often this first move is speculative and in the worst case the agent may forfeit the chance of achieving a full score. Solving puzzles, such as Peg, that have sub-goals that guide the agent toward higher scores is often easier in real-time. Other puzzles have misleading sub-goals and are thus harder to solve in such settings. Our implementation favors the known over the unknown and chooses the highest score observed over hoping for optimality with no guarantees.

Chapter 6

Conclusions

The goal of this project was to analyze and address issues that our GGP agent had with single-role puzzles. Many single-role puzzles are unsuitable for the sampling-based UCT method due to the fact that informative feedback is scarce. By building upon the work done for general purpose planners we implemented a system that derives a heuristic by solving a simplified version of the original puzzle and uses that solution as an estimate of the true solution length.

Like the *Progression* based relaxations, we must by default solve the relaxed puzzle every time a heuristic is derived. As this is the most time consuming part of the process we proposed and implemented a method based on a lookup cache similar to HSPr. However, because of how hard it is to define the *Regression Space* for game descriptions in GDL, we cannot implement the full one pass lookup that is used by HSPr, but instead we use an approximation.

When an unestimated proposition is encountered a relaxation is performed and the proposition cache is updated so that the lowest derived estimate for each proposition is stored in the cache. However, if every proposition in a state has a cached value the state can be estimated from cache without performing the relaxation process. Several relaxations are required for the cached values to converge to their true distance value and until that happens the heuristic is not guaranteed to be admissible. To ensure convergence we enforce relaxations even when a cached value is available for every proposition in the state. The threshold we call k determines how many cache hits are allowed in a row before enforcing a relaxation. We also tested an algorithm where the frequency of forced relaxations was adaptive in that it was incremented or decremented based on whether the cached estimate and the relaxation estimate agreed or not.

As the relaxation process is the inhibiting factor in the search progress we would like to do as few relaxations as possible and allow many cache hits as long as the informativeness is not compromised. The adaptive k threshold is a promising idea but it does run into problems with puzzles where cache hits and relaxation estimates agree frequently but then overestimate a few crucial states that will lead to a solution. The overestimated good states will not get expanded due to their overestimate and are essentially lost on the open list causing the search to thrash around for a long time before finally expanding them. However, for many puzzles the caching variants perform quite well.

Unlike the planning competitions, where the planner tries to come up with a good plan as quickly as possible, the GGP competitions require the agent to perform actions in real-time, that is, once the play clock is up the agent must perform a legal action. What this means is that a GGP agent cannot automatically assume all actions to be reversible or that they can be undone. In some puzzles the agent can forgo an optimal score by failing to choose the correct first action. Hence the assumption that can safely be made by planners, that the puzzle is solvable from any state, only holds for the initial state in GGP. Once the agent has set out on its journey, no matter how much time is allocated, there is no guarantee it can achieve an optimal score.

Because of the importance of choosing the correct action right from the start we choose to apply a complete search algorithm. Our algorithm is called Time-Bounded and Injection-based A* (TBIA*) and is based on Time-Bounded A* (TBA*). TBA* enhances the classical A* algorithm to work in real-time by interleaving planning and execution. In addition it maintains two paths, the best path at each time step called *pathNew* and a path to the best achievable goal called *pathFollow*. The agent will traverse along *pathFollow* until a better path has been discovered. The main difference between TBA* and TBAI* lies in how *pathFollow* is updated. In TBAI*, when new preferred path is found, instead of backtracking like TBA* does, we prefer to reset the search effort and have the agent search its way onto better paths. This way puzzles with irreversible moves can be handled. To facilitate the rediscovery of previously unusable paths at a later time we inject low heuristic values to the states of unusable but good paths.

We have shown that the methods described above can solve puzzles that can be considered unsolvable by the simulation based methods, such as the 8-puzzle. We have also shown that the performance of our methods depends heavily on how the puzzle description is formulated. The main drawback of the current implementation is a lack of flexibility towards dealing with such diversity.

Bibliography

Bacchaus, F. (2001). The AIPS 2000 Planning Competition. *The AI Magazine*, *22*(3), 47–56.

Björnsson, Y., Bulitko, V., & Sturtevant, N. (2009). TBA*: Time-Bounded A*. In *Proceedings of the 2009 International Joint Conferences on Artificial Intelligence.* (In Press)

Blum, A., & Furst, M. (1995). Fast Planning Through Planning Graph Analysis. *Artificial Intelligence*, *90*, 1636–1642.

Bonet, B., & Geffner, H. (2001). Planning as Heuristic Search. *Artificial Intelligence, Special issue heuristic Search*, *129*, 5–33.

Fikes, R., & Nilsson, N. (1971). STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *Artificial Intelligence*, *1*, 27–120.

Finnsson, H. (2007). CADIA-Player: A General Game Playing Agent (Master's thesis, School of Computer Science). Reykjavik University.

Haslum, P. (2006). Improving Heuristics Through Relaxed Search – An Analysis of TP4 and HSP*a in the 2004 Planning Competition. *Journal of Artificial Intelligence Research*, *25*, 233–267.

Haslum, P., & Geffner, H. (2000). Admissible Heuristics for Optimal Planning. In *Proceedings of the 5th International Conference on AI Planning and Scheduling* (pp. 140–149).

Helmert, M. (2003). Complexity results for Standard Benchmark Domains in Planning. *Artificial Intelligence*, *143*, 219–262.

Hoffman, J. (2005). Where "Ignoring Delete Lists" Works: Local Search Topology in Planning Benchmarks. *Journal of Artificial Intelligence Research*, *24*, 685–758.

Hoffman, J., & Nebel, B. (2001). The FF Planning System: Fast Plan Generation Through Heuristic Search. *Journal of Artificial Intelligence Research*, *14*, 253–302.

Kambhampati, S., Parker, E., & Lambrecht, E. (1997). Understanding and extending GraphPlan. In *Proceedings of the 4th European Conference on Planning*.

Kocsis, L., & Szepesvári, C. (2006). Bandit Based Monte-Carlo Planning. In *Proceedings of the 17th European Conference on Machine Learning* (pp. 282–293).

Long, D., & Fox, M. (2003). The 3rd International Planning Competition: Results and Analysis. *Journal of Artificial Intelligence Research*, *20*, 1–59.

Love, N., Genesereth, M., & Hinrichs, T. (2006). *General Game Playing: Game Description Language Specification* (Tech. Rep. No. LG-2006-01). Stanford University.

McDermott, D. (1997). *PDDL - The Planning Domain Definition Language.* Available at http://ftp.cs.yale.edu/pub/mcdermott.

Pednault, E. (1989). ADL: Exploring the middle ground between STRIPS and the situation calculus. In *Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning* (pp. 324–332).

REYKJAVÍK UNIVERSITY

HÁSKÓLINN Í REYKJAVÍK

School of Computer Science
Reykjavík University
Kringlan 1, IS-103 Reykjavík, Iceland
Tel: +354 599 6200
Fax: +354 599 6301
http://www.ru.is